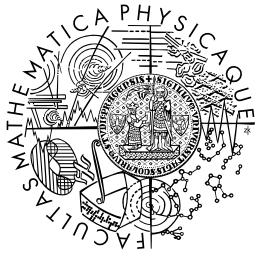

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

RNDr. Ing. Otakar Trunda

Heuristic Learning for Domain-independent Planning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the doctoral thesis:
prof. RNDr. Roman Barták, Ph.D.

Study programme:
Computer science

Study branch:
Theoretical Computer Science and Artificial Intelligence

Prague 2022

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Rád bych tuto práci věnoval své mamince, která mě motivovala ke studiu a společně s tátou mě po celou dobu podporovali.

Děkuji své přítelkyni Kamile za její trpělivost a vytváření příjemného prostředí, kde bylo možné skloubit studijní povinnosti s pracovním a rodinným životem.

Dále chci poděkovat profesoru Bartákovi za vedení této práce a za všechny znalosti, které mi předal během naší mnohaleté spolupráce.

Title: Heuristic Learning for Domain-independent Planning

Author: RNDr. Ing. Otakar Trunda

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract:

Automated planning deals with the problem of finding a sequence of actions leading from a given state to a desired state, e.g., solving Rubik's Cube, delivering parcels, etc. The state-of-the-art automated planning techniques exploit informed forward search guided by a heuristic, where the heuristic estimates a distance from a state to a goal state.

In this thesis, we present a technique to automatically construct an efficient heuristic for a given planning domain. The proposed approach is based on training a deep neural network using a set of previously solved planning problems from the same domain. We use a novel way of extracting features for states which doesn't depend on usage of existing heuristics. The trained network can be used as a heuristic on any problem from the domain of interest without any limitation on the problem size. Our experiments show that the technique is competitive with popular domain-independent heuristic.

We also introduce a theoretical framework to formally analyze behavior of learned heuristics. We state and prove several theorems that establish bounds on the worst-case performance of learned heuristics.

Keywords: Heuristic learning, Machine learning, Classical planning, Heuristic search

Název práce: Učení heuristik pro doménově nezávislé plánování

Autor: RNDr. Ing. Otakar Trunda

Katedra teoretické informatiky a matematické logiky

Vedoucí disertační práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt:

Automatizované plánování se zabývá hledáním posloupnosti akcí, které vedou k dosažení cílového stavu ze zadaného počátečního stavu, např. řešení Rubikovy kostky, doručování balíků, atd. Moderní plánovací techniky jsou založené na informovaném dopředném prohledávání řízeném heuristikou, kde heuristika poskytuje odhad vzdálenosti daného stavu od cílového stavu.

V této práci představujeme techniky pro automatické vytvoření efektivní heuristiky pro jakoukoli zadanou plánovací doménu. Navržené řešení je založené na trénování hluboké neuronové sítě s využitím dříve vyřešených plánovacích problémů ze stejné domény. Navrhli jsme nový způsob extrakce příznaků pro stavy plánovacích problémů, která není závislá na využití existujících heuristik. Natrénovanou síť je možné využít jako heuristiku při řešení jakéhokoli problému z dané domény bez ohledu na velikost problému. Experimenty ukazují, že navržená technika je kompetitivní s populární doménově nezávislou heuristikou.

Představujeme také teoretický rámec pro formální analýzu vlastností naučených heuristik. Formulujeme a dokazujeme věty, které stanovují meze na výkonnost naučených heuristik v nejhorším případě.

Klíčová slova: Učení heuristik, Strojové učení, Klasické plánování, Prohledávání s heuristikou

Contents

Introduction

Deterministic sequential decision-making problems, such as Rubik's cube, Sokoban or Vehicle Routing are among the hardest combinatorial optimization tasks. Currently, the most popular approach for solving them is breadth-first search in the state space of the problem together with a heuristic distance-estimator to guide the search, i.e. A^* algorithm and its variants. Performance of the algorithm heavily depends on the heuristic used.

In this thesis, we use machine learning techniques to improve performance of such algorithms. We present a way to automatically construct a strong heuristic for any given problem. In the literature, this task is called *Heuristic learning* (HL). We investigate two main approaches:

1. Improving existing heuristics by automatically identifying their weaknesses and overcoming them.

2. Creating strong heuristics automatically from scratch.

Motivation

Machine learning (ML) is currently revolutionizing many areas of computer science as well as industries[1]. In the fields of sound processing ([63]), image processing ([89]) and language processing ([90]), most of the traditional human-designed algorithms have now been replaced by deep neural networks. The same revolution occurred in game-playing as well. Deep neural network is a key component of *AlphaGo* engine - the first computer program ever to beat a top level human player in Go[2]. The similar technique was later used in *DeepStack* - currently the best Texas Hold'em Poker engine ([64]) and *AlphaZero* which can play chess, Go and shogi at a super-human level ([83]).

Traditional chess engines have been able to beat best human players for some time now. They are based on the alpha-beta algorithm and they utilize expert knowledge about the game acquired by human players during the past thousand years. Nowadays, *AlphaZero* can beat them after just few days of learning from self-play, without any human knowledge incorporated. After observing games played by *AlphaZero*, chess grand masters have even been forced to reconsider several principles of chess strategy that were widely believed to be valid in the past[3].

Another version of the learning engine called *AlphaStar*[4] is currently trying to master the popular video game Starcraft II. It is already able to compete with top level human players and keeps improving[5]

[1] https://www.artiba.org/blog/real-world-examples-how-ai-is-revolutionizing-top-industries

[2] https://www.cbc.ca/news/science/go-google-alphago-lee-sedol-deepmind-1.3488913

[3] https://www.technologyreview.com/2017/12/08/147199/alpha-zeros-alien-chess-shows-the-power-and-the-peculiarity-of-ai/

[4] https://en.wikipedia.org/wiki/AlphaStar_(software)

[5] https://www.theverge.com/2019/10/30/20939147/deepmind-google-alphastar-starcraft-2-research-grandmaster-level

These successes motivate us to use ML in the field of automated planning as well. ML has been used for planning already but we can hardly talk about a revolution. Not yet at least. One of the biggest successes so far would be mastering Atari games in [59, 15].

In the field of classical planning, vast majority of techniques is still based on hand coded tools. We believe that there is a great potential in using ML in informed forward search, especially in heuristic design. Planning can be viewed as a special case of game - a single player game - so in principle it should be possible to apply similar techniques here as well.

Data-driven approaches to combinatorial optimization are currently a hot research topic. Among workshops that specialize on this area, we can mention

- Bridging the Gap Between AI Planning and Reinforcement Learning[6]
- Knowledge Representation & Reasoning Meets Machine Learning[7]
- Data Science Meets Optimisation[8]
- and others

Goals

Our goal is to develop a domain-independent heuristic-learning system, i.e., to apply machine learning to automatically construct strong heuristic for any given domain. We work with classical STRIPS planning and we use a supervised learning paradigm.

The intended use-case of our technique is as follows. We are given a large set of planning problems of the same type (i.e. same *domain*) and a potentially infinite amount of time to analyze them. During the analysis phase, the system automatically extracts useful knowledge about the domain. After the analysis, the system will be able to solve new, previously unseen problems from the same domain quickly by utilizing the knowledge previously obtained. The knowledge is represented in the form of a heuristic function that A^* algorithm can use.

Time required for the analysis phase is not taken into account. Our goal is to show that strong heuristic **can** in principle be learned without any expert knowledge and that it can outperform human-designed heuristics.

Structure

The thesis is structured as follows. The first chapter provides an introduction to automated planning, machine learning and especially heuristic learning together with related papers. In the second chapter, we study properties of heuristics with respect to heuristic learning and we propose several ways of improving existing heuristics.

[6] https://prl-theworkshop.github.io/

[7] https://kr2ml.github.io/

[8] https://tailor-network.eu/events/ijcai-2022-dso-workshop-data-science-meets-optimisation/

The third chapter presents our HL framework and in the fourth we describe our novel way of extracting features. The fifth chapter contains an experimental evaluation of the proposed techniques. Our method outperforms a popular human-designed tool in the number of problems solved as well as solution quality on two hard planning domains.

The last chapter provides theoretical results about efficiency of our method. We show that under some reasonable assumptions, the learned heuristic will find high quality solution with some high probability.

1. Background and Related Work

In this chapter, we describe the problem of action planning as well as introduce the fields of machine learning and heuristic learning. We also review related works and establish notation.

1.1 Automated Planning

Planning deals with finding a sequence of decisions (actions) which - when executed in the given environment - leads to achieving the given goal. Executing actions affects state of the environment. We work with *classical planning* ([65]) which specifies the previous description in several ways:

1. the environment is fully observable, deterministic and static
2. executing actions is instantaneous, i.e. time is not taken into account
3. goal conditions are explicit, i.e. a set of goal states is given and the task is just to reach any of those states
4. the solver is *domain-independent*, in a sense that it accepts its inputs in a standardized format and is applicable to a wide range of different environments

More formally:

Definition 1 (Classical planning task)**.** ***Classical planning task*** *T is a tuple $T = (S, A, \gamma, s_0, goal, c)$, where*

- *S is a set of states*
- *A is a set of actions*
- *$\gamma : S \times A \mapsto S$ is a partially defined successor function*
- *$s_0 \in S$ is the initial state*
- *$goal : S \mapsto \{0, 1\}$ is a goal condition and $\{s \in S \mid goal(s) = 1\}$ is a set of all goal states*
- *$c : A \mapsto \mathbb{R}$ is a cost function*

We will also use the term *planning problem* when referring to planning tasks. If P is a planning problem then by S^P we denote the set of states of P, by A^P the set of actions of P and so on. When the problem P is not important or clear from context, we will just write S, A, γ and so on.

γ is a partial function, i.e. it need not be defined for all elements of $S \times A$. When $\gamma(s, a)$ is defined, we say that action a is *applicable* to state s. Otherwise we say that it is *not applicable*.

Example 1.1

Consider the following scenario. There are two houses denoted *Purple house* and *Yellow house*, a van and a parcel. The parcel is currently located at the Yellow house and the goal is to transport it to the Purple house. It is possible to move the van from one house to the other and to load or unload the parcel.

We can model the scenario as a simple planning problem **P**. The set of states **S** in this case contains 6 elements: $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ such that s_1 represents the situation where both the van and the parcel are near the Yellow house, s_2 is the same but the parcel is loaded in the van, in s_3 the parcel in near the Purple house while the van is at the Yellow house, s_4 is the other way around: van at the Purple house while parcel at the Yellow house, in s_5 both object are at the Purple house and the parcel is loaded and s_6 is the same except the parcel is not loaded. That covers all possible situations in our scenario.

We have 6 actions here: $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$. a_1 represents driving the van from the Yellow house to the Purple house, a_2 represents driving the other way, a_3 represents loading the parcel at the Yellow house and a_4 unloading it at the Yellow house. Actions a_5 and a_6 represent loading and unloading the parcel at the Purple house.

The successor function is defined such that the parcel can only by loaded if it is at the same position as the van. When unloaded, the parcel will be at the same position as the van and moving the van changes its position accordingly. E.g., action a_3 is applicable to state s_1 and result of application is state s_2, i.e. $\gamma(s_1, a_3) = s_2$. On the other hand, a_3 is not applicable to s_4 therefore $\gamma(s_4, a_3)$ is not defined.

The whole state space is depicted in figure 1.1 and its simplified diagram in figure 1.2.

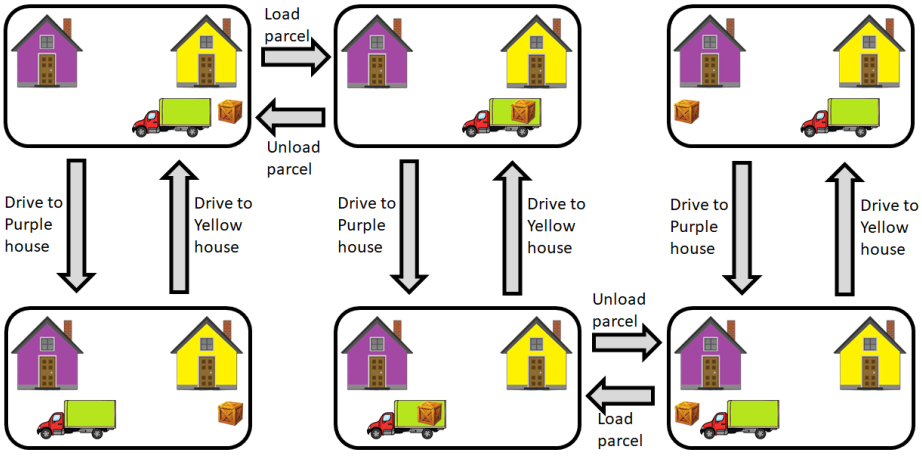

Figure 1.1: State-space of a simple planning problem.

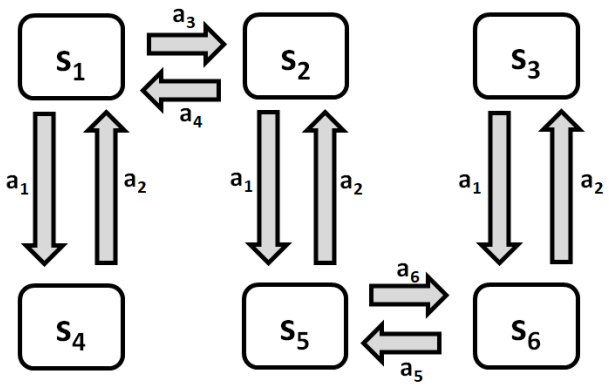

Figure 1.2: Simplified diagram of the state-space.

The initial state is s_1, there are two goal states s_3 and s_6 and all actions have the same cost, i.e. $\forall a \in A : c(a) = 1$.

Definition 2 (Set of finite sequences)**.** *Let X be a non-empty set. By $X^{<\infty}$ we denote the* ***set of all finite non-empty sequences*** *of elements of X.*

Definition 3 (Extended successor function)**.** *Given a problem $P = (S, A, \gamma, s_0, goal, c)$, we define the* ***extended successor function*** *$\gamma_+^P : S \times A^{<\infty} \mapsto S$ as*

- $\gamma_+(s, \langle a \rangle) = \gamma(s, a)$
- $\gamma_+(s, \langle a_1, a_2, \ldots, a_n \rangle) = \gamma_+(\gamma(s, a_1), \langle a_2, a_3, \ldots, a_n \rangle)$

Definition 4 (Extended cost function)**.** *We also define* ***extended cost function*** *$c_+^P : A^{<\infty} \mapsto \mathbb{R}$ as $c_+^P(\langle a_1, a_2, \ldots, a_n \rangle) = \sum_{i=1}^{n} c(a_i)$.*

Definition 5 (Reachable state)**.** *We say that state $s \in S$ is* ***reachable*** *if there exists a sequence of actions $\pi \in A^{<\infty}$ such that $\gamma^+(s_0, \pi) = s$ where s_0 is the initial state.*

Definition 6 (Plan)**.** *A sequence of actions $\pi \in A^{<\infty}$ is called* ***plan for*** *P if $\gamma_+^P(s_0^P, \pi)$ is a goal state. The value $c_+^P(\pi)$ is called* ***cost*** *of plan π.*

We do not work with partially ordered planning so *plan* will always be a fully-ordered sequence.

Example 1.2

In the previous example, a sequence $\langle a_3, a_1, a_6 \rangle$ is a plan. Cost of the plan is 3.

Now we can define *solution* to a planning problem. There are two alternatives: *optimal planning* deals with finding a *plan* for P whose cost is minimal among all plans - i.e. an *optimal plan*. Only optimal plans are considered *solutions* in this scenario.

Satisficing planning, on the other hand, just aims at finding *some* plan. In this scenario, we accept any plan as a solution but we still care about quality of the plan. Plans with lower cost are considered better.

Literature distinguishes between planning *with action costs* and *without action costs*. In the latter case, costs of all actions are the same, so the task is just to minimize the number of actions in the plan. This scenario is sometimes referred to as *planning with unit costs* and the term *classical planning* is typically connected with the unit costs scenario.

In this thesis, we focus on domains with unit action costs. Our technique is applicable also to non-unit costs scenarios to some degree. See 1.3.2 for details.

There are currently two major formalisms used to describe planning tasks: the *STRIPS* formalism and the *FDR* formalism. We will be using both of them so we define them here.

1.1.1 STRIPS

STRIPS (STanford Research Institute Problem Solver. See [26]) is based on predicate logic. It uses *constants* and *predicates* to describe state-space of the planning problem. Constants typically correspond to objects in the world, while predicates describe their relations.

States

We are given a set of constants C, and a set of predicate symbols Ψ. Each predicate symbol has its arity $ar : \Psi \mapsto \mathbb{N}^0$. A set Q of all *instantiated predicates* is defined as $Q = \{(p, c_1, c_2, \ldots, c_{ar(p)}) \mid p \in \Psi, c_i \in C\}$. A *state* is then defined as an assignment of values $\{true, false\}$ to all predicates of Q, i.e. $s : Q \mapsto \{true, false\}$. A set of all states denoted by S contains all such assignments, hence $|S| = 2^{|Q|}$.

For a state $s \in S$ and a predicate $q \in Q$, we say that q *holds* in the state s or that it is *positive* in the state, if $s(q) = true$. Otherwise we say that predicate q *doesn't hold* in s or that it is *negative* in the state. A set of all predicates that are positive in s is denoted by s^+ and the set of all negative predicates by s^-. STRIPS states are typically represented by a set of positive predicates. Predicates that are not explicitly mentioned as positive are considered negative. Typically, vast majority of predicates are negative in all states.

Unary predicates can model state of an object, e.g. *isEmpty(hoist1)* or can represent types of objects, e.g. *isCity(city2)*. Nullary predicates can represent global binary features, like *isDaytime*. Binary predicates allows to model relations between two objects, e.g. *at(robot1, location1)*. Ternary or general n-ary predicates can be used to represent more complex relations.

Example 1.3

> Let's model the previous example using the STRIPS formalism. We will use 4 constants: *van*, *parcel*, $PHouse$ and $YHouse$ and two predicate symbols *at* and *in* both with arity of 2. In the initial state there are 2 positive predicates: $at(van, YHouse)$ and $at(parcel, YHouse)$. All other predicates are negative.

Goal condition

A *partial state* is an assignment $s : Q \mapsto \{true, false, *\}$ where the asterisk represents a *"don't care"* value. States are considered special cases of partial states.

Given two partial states s_1, s_2, we say that s_1 is *subsumed* by s_2 if $\forall q \in Q : s_2(q) = *$, or $s_1(q) = s_2(q)$.

Goal condition is specified via a partial state s_g. All states that are subsumed by s_g are considered goal states.

Example 1.4

> In our example, the goal condition is $at(parcel, PHouse)$. Any state in which this predicate holds is considered a goal state. More formally:
> $s_g(at(parcel, PHouse)) = true$ and $\forall q \in Q \setminus \{at(parcel, PHouse)\} : s_g(q) = *$.

Transitions

Transitions are defined in a form of so called *actions*. A STRIPS action is a tuple $a = (prec^+, prec^-, \mathit{eff}^+, \mathit{eff}^-)$, where $prec^+, prec^-, \mathit{eff}^+, \mathit{eff}^- \subseteq Q$. Action $a = (prec^+, prec^-, \mathit{eff}^+, \mathit{eff}^-)$ is applicable to a state $s \in S$, if $prec^+ \subseteq s^+$ and $prec^- \subseteq s^-$. If action a is applicable to s then successor of s via a is a state s' defined as follows:

- $\forall q \in \mathit{eff}^- : s'(q) = false$
- $\forall q \in \mathit{eff}^+ \setminus \mathit{eff}^- : s'(q) = true$
- $\forall q \in Q \setminus \mathit{eff}^+ \setminus \mathit{eff}^- : s'(q) = s(q)$

If action a is not applicable to s then the transition is not defined.

Example 1.5

> Actions from our example can be modeled as follows:
>
> - $a_1 = (\{at(van, YHouse)\}, \emptyset, \{at(van, PHouse)\}, \{at(van, YHouse)\})$
> - $a_2 = (\{at(van, PHouse)\}, \emptyset, \{at(van, YHouse)\}, \{at(van, PHouse)\})$
> - $a_3 = (\{at(van, YHouse), at(parcel, YHouse)\}, \emptyset,$ $\{in(parcel, van)\}, \{at(parcel, YHouse)\})$
> - $a_4 = (\{at(van, YHouse), in(parcel, van)\}, \emptyset,$ $\{at(parcel, YHouse)\}, \{in(parcel, van)\})$
> - $a_5 = (\{at(van, PHouse), at(parcel, PHouse)\}, \emptyset,$ $\{in(parcel, van)\}, \{at(parcel, PHouse)\})$
> - $a_6 = (\{at(van, PHouse), in(parcel, van)\}, \emptyset,$ $\{at(parcel, PHouse)\}, \{in(parcel, van)\})$

Fluent and rigid predicates

It is useful to distinguish between two types of predicates: *rigid* ones and *fluent* ones.

Definition 7 (Fluent and rigid predicates)**.** *Predicate $q \in Q$ is called **rigid** if it is never present among the effects (positive or negative) of any action. If there is a predicate symbol $\psi \in \Psi$ such that all predicates q_i that use ψ are rigid, we say that the predicate symbol ψ is **rigid**. Predicates and predicate symbols that are not rigid are called **fluent**.*

Rigid predicates occur in preconditions of actions but they never change their value during the planning process. They will be positive in a reachable state if any only if they are positive in the initial state hence their values don't need to be stored in each state separately (like the fluent ones) but rather just once for all reachable states.

Example 1.6

Rigid predicates are used to describe static properties of the problem, like a road map. E.g. there could be predicate $adjacent(a, b)$ which signifies that it is possible to drive from a to b directly.

They are also often used to specify *types* of objects. E.g. in the Zenotravel domain (see A.1.1) there is a unary predicate symbol *city* and in the initial state we can find predicates $city(city1), city(city2)$, etc.

1.1.2 Finite Domain Representation (FDR)

Finite Domain Representation (also called *Multi-valued Planning Task*) uses multi-valued state variables to describe states (see [43]). It is similar to the predicate formalism and extends it in some respects. It is based on *state variables* instead of instantiated predicates and it allows these variables to be assigned values from a larger domain than just $\{true, false\}$.

A finite set of variables $V = \{v_1, v_2, \ldots, v_n\}$ is given and for each $v \in V$ we are given its domain $\mathcal{D}(v)$.

A state s is an assignment $s : V \mapsto \Pi_{i=1}^{n} \mathcal{D}(v_i)$ such that
$\forall v_i \in V : s(v_i) \in \mathcal{D}(v_i)$. The number of states in this case is $|S| = \Pi_{i=1}^{n} |\mathcal{D}(v_i)|$.

Partial states, goal condition and actions are defined in the same way as in STRIPS, here we just use V instead of Q and $\mathcal{D}(v_i)$ instead of $\{true, false\}$. As we don't distinguish between *positive* and *negative* predicates, actions just have their preconditions and effects defined as partial states. We will write action using notation $a = (preconditions \mapsto effects)$.

Example 1.7

To model our previous example in FDR, we need just two variables *van_Loc* and *parcel_Loc*. Variable *van_Loc* will represent location of the van and it will have domain $\mathcal{D}(van\_Loc) = \{YHouse, PHouse\}$. Variable *parcel_Loc* will capture location of the parcel with three possible values $\{YHouse, PHouse, van\}$.

Actions in this case look as follows.

- $a_1 = (van\_Loc = YHouse \mapsto van\_Loc = PHouse)$
- $a_2 = (van\_Loc = PHouse \mapsto van\_Loc = YHouse)$
- $a_3 = (van\_Loc = YHouse, parcel\_Loc = YHouse \mapsto parcel\_Loc = van)$
- $a_4 = (van\_Loc = YHouse, parcel\_Loc = van \mapsto parcel\_Loc = YHouse)$
- $a_5 = (van\_Loc = PHouse, parcel\_Loc = PHouse \mapsto parcel\_Loc = van)$
- $a_6 = (van\_Loc = PHouse, parcel\_Loc = van \mapsto parcel\_Loc = PHouse)$

1.1.3 Languages

Planning tasks need to be written down using a specific *language* that planners understand and can work with. The most popular language nowadays is *PDDL - Planning Domain Definition Language* ([38]). There are several variants of the language capable of describing classical *STRIPS* problems as well as more complex planning tasks like *ADL (Action Description Language)* that allows arbitrary logic formulas in preconditions and goals, probabilistic planning and others.

A PDDL description of a STRIPS planning task is divided into two files: a *Domain* file and a *Problem* file. The domain file contains description of predicate symbols (including types) and action templates called *operators*. The problem file then contains names of objects (i.e., constants) and instantiated predicates that describe both initial and goal state. A *Planning domain* is a set of all planning tasks that share the same domain file. It represents a class of similar problems.

Example 1.8

In case of Zenotravel (see A.1.1), the domain file specifies that there are *planes*, *cities* and *passengers*, passengers can be either at cities or in planes, planes can fly between cities and passengers can embark on and disembark from planes.

The problem file then specifies the number of cities and their identifications as well as the number and identifications of passengers and planes, their current locations and their destinations.

Multi-valued planning tasks, on the other hand are typically specified using the SAS^+ language (Simplified Action Structures). It just lists the number of variables and for each variable the size of its range. Variables are encoded as numbers from 0 to $|V| - 1$ and domain of each variable v_i is encoded into a set $\{0, 1, \ldots, |\mathcal{D}(v_i)| - 1\}$. A state is then encoded simply as an array of integers with size of $|V|$.

SAS^+ format doesn't use action templates (operators) but rather enumerates all instantiated action straightaway. Also, it doesn't separate information about *domain* and *problem* to two files but uses a single file instead. See *https://www.fast-downward.org/TranslatorOutputFormat* for more details about the SAS^+ format.

1.1.4 Comparison and translation between formats

SAS^+ uses more concise inner representation of states and it is therefore better suited for implementation of most state-space search algorithms. States can be

represented simply as arrays of integers. PDDL on the other hand can provide better insight into the structure of the problem. Also, PDDL description can be more efficient for very large problems as it uses *operators* to define actions. Operators are templates that describe how to create actions. In SAS^+ format, all actions have to be enumerated while PDDL only provides operators and hence SAS^+ input file could be exponentially larger than the PDDL one.

A PDDL problem can be directly translated to SAS^+ by creating a single binary variable for each instantiated predicate. SAS^+ however often allows us to create much more efficient encoding.

Definition 8 (MutEx)**.** *We say that two predicates are* ***mutually exclusive*** *(MutEx) if they can never both be positive in any reachable state. A* ***mutex group*** *is a set of predicates such that every pair of them is MutEx.*

Example 1.9

> In our example, a set of predicates
> $\{at(parcel, PHouse), at(parcel, YHouse), in(parcel, van)\}$ forms a mutex group.

Many of such mutex groups can be discovered automatically by analyzing the initial state and structure of actions. Each mutex group can then be modeled using a single SAS^+ variable which often generates much smaller state space than the original PDDL representation.

There is a translator available ([43]) that translates any PDDL problem to a concise SAS^+ representation in a reasonable time. We use PDDL format for the domain analysis and SAS^+ representation for the actual planning.

1.1.5 Popular planning techniques

Currently, the most popular technique for solving planning problems is informed forward search in the state-space of the problem, i.e. a variant of A^* or IDA^* algorithm [1] ([20, 77]). Those informed algorithms make use of *heuristic distance estimators* or simply *heuristics* to guide the search.

Heuristics

Definition 9 (Heuristic)**.** *Given a planning problem* $P = (S, A, \gamma, s_0, goal, c)$*,* ***heuristic*** h *is a mapping* $h : S \mapsto \mathbb{R}_0^+$ [2]*.*

Heuristics are the key components of currently used search algorithms and a lot o research effort now focuses on developing accurate and easy-to-compute heuristics[3]. In general, heuristics can be divided into two categories: **domain-specific heuristics** and **domain-independent heuristics**.

[1] Planners based on this paradigm have been stably winning the International Planning Competitions (IPCs) for the past 10 years. See *https://ipc2018-classical.bitbucket.io/* for information on the latest IPC and *https://ipc2018-classical.bitbucket.io/planner-abstracts/ipc-2018-planner-abstracts-classical-tracks.pdf* for abstracts of participating planners.

[2] In some specific scenarios, heuristics that produce negative values are also used, e.g. in *cost partitionings* ([55]).

[3] See for example https://www.aaai.org/Library/ICAPS/icaps-library.php

Domain independent heuristics work with the PDDL or SAS^+ description of the problem and are in principle applicable to any problem defined in the respective formalism although performance of the heuristic often varies based on type of problem. Some heuristics focus on problems with actions costs, some of them were developed specifically for unit-costs problems and some can be used in both scenarios.

Domain-specific heuristics, one the other hand, are only applicable to a single domain or a small set of similar domains. They can leverage domain specific knowledge and hence are typically more efficient than domain-independent ones. Developing a domain-specific heuristic for a new domain, however requires a significant effort and access to expert knowledge while domain-independent heuristics can be used off-the-shelf. As an example of domain-specific heuristic, we can look at Sokoban solvers, e.g. [70, 69].

Among popular domain-independent heuristics we can mention $h_{GoalCount}$ and *critical path* heuristics $h_0, h_1, \ldots, h_k$ ([65]), *delete relaxation* heuristics like h_{FF} ([48]), h_{CFF} ([25]) or $h_{Red-Black}$ ([53]), *landmarks*-based heuristics like h_{LM-cut} ([44, 6, 96]), *abstraction* heuristics like Pattern databases ([75, 82]), Merge-and-Shrink ([47, 7]) and Cost partitioning ([28, 78]). Other approaches also exist, e.g. operator counting ([71]).

We will describe two heuristics here in greater detail as we will be using them in our experiments.

GoalCount heuristic - h_{GC} is a very simple heuristic, one of the first ever created for the purposes of planning. It counts the number of goal predicates that are not yet satisfied in current state, i.e. $h_{GC}(s^P) = |G^P \setminus s^P|$. The heuristic is admissible as long as no action can accomplish multiple goal predicates simultaneously.

FastForward heuristic - h_{FF} ([48]) is a much more sophisticated one. It is based on so called *delete relaxation*: After applying an action, some predicates become positive while others become negative (i.e. they are *deleted* from the state). The delete relaxation does not delete those predicates but instead marks them as *don't care*, i.e. from that point on, they will pretend to have whatever truth value is required of them.

Given a state s, it is possible to compute a *relaxed plan* - a plan from s using the delete relaxation mechanism - and use the length of such plan as a lower bound on the length of the actual plan. Since computing relaxed plans is still hard, h_{FF} uses an approximation of them and hence is not guaranteed to be admissible. (See definition 13.)

The *Fast Forward* planning system won the International Planning Competition (IPC) in 2000 and performed well also in the following IPCs. Since then, h_{FF} has become a popular benchmark to test new heuristics against.

The idea of *delete relaxation* has been later generalized to the *Red-Black planning* ([16]) and is still being extensively developed and used in heuristics, e.g. in [54, 25]. Also, h_{FF} is the default heuristic in *Fast Downward* ([41, 77]) - currently one of the most popular and best performing planning systems. See *http://www.fast-downward.org/*.

Search algorithms

As for the search algorithm, planners are typically based on standard A* or IDA* or their modified versions adjusted specifically for planning. Among those modifications, we can mention *multiple open lists* and *helpful actions* ([48, 41]), *control rules*, usage of *portfolios* of algorithms and/or heuristics ([46]), and others ([65]). There are currently many versions of the A^* algorithm tailored for specific types of problems, e.g. [40].

These diverse planning approaches are often intended for specific use-case scenarios. For example, some techniques guarantee finding optimal plans. Others do not but they specialize in finding high-quality plans in reasonable time. Other techniques only focus on finding just any plan as quickly as possible without taking its cost into account. See e.g. *https://ipc2018-classical.bitbucket.io/#tracks* for examples of usage-scenarios.

Techniques also exist that don't rely on state-space forward search like *HTN planning* ([65]), symbolic search ([19, 57, 21]), or *plan-space planning* ([65, 18]).

Other planning-related problems also exist like proving non-existence of a plan ([22]), plan recognition ([62]), planning in partially observable environments, in non-deterministic environments, etc. ([66]).

In this thesis, we deal only with the problem of finding plans and we work with informed forward search algorithms.

1.1.6 Notation

In the rest of this thesis we use the following notation.

- P, P_1, P_i denote planning problems
- S or S^P denote set of all states of some planning problem
- s, s^P, s_1, s_i denote states of some planning problems
- s_0, s_0^P denote initial state of some planning problem
- given a reachable state s of some problem P, $g(s)$ denotes cost of the cheapest path (plan) from s_0^P to s.

Definition 10 (Goal-distance)**.** *Let $s^P \in S^P$ be state. By $h^*(s^P)$ we denote **the goal-distance** of s^P in P, i.e. the cost of the optimal plan from s^P to some goal state of P, or ∞ if there is no path from s^P to a goal state. We refer to $h^*(s)$ as to **goal-distance**, **distance-to-go**, **cost-to-go** or **perfect heuristic**.*

1.2 Machine Learning

Machine learning (ML) is a wide field which nowadays encompasses a large variety of different techniques. We will only introduce here the area of supervised learning as that is the one we are using in our framework. We will first introduce several notions from statistics and ML.

1.2.1 Notation

Throughout this text we use the following notation:

- *Random variables* (RVs) are denoted by capital roman letters such as $\mathcal{Z}$
- *Probability* of an event ω is denoted by $\mathsf{P}[\omega]$, e.g. $\mathsf{P}[\mathcal{Z} < a]$
- *Expected value* of a RV $\mathcal{Z}$ is denoted by $\mathsf{E}[\mathcal{Z}]$
- *Variance* of a RV $\mathcal{Z}$ is denoted by $\mathrm{var}[\mathcal{Z}]$
- By $\mathcal{Z} \sim N(\mu, \sigma^2)$ we denote the fact that RV $\mathcal{Z}$ is normally distributed with mean μ and variance σ^2
- By $\exp\{x\}$ we denote e^x
- *Tensor space* (over $\mathbb{R}$) is a set $\mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \times \cdots \times \mathbb{R}^{d_k}$ for some $d_1, d_2, \ldots d_k \in \mathbb{N}$.
- *Tensor* is element of a tensor space

Tensors are higher-dimensional matrices. Vectors and matrices are both special cases of tensors, vectors are 1-dimensional tensors while matrices are 2-dimensional ones. In the following text, we will be using the term *vector* instead of tensor. I.e., by *vector* we actually mean either *vector*, *matrix* or general *tensor*. When saying that two vectors have the same size, we mean that those tensors come from the same tensor space.

1.2.2 Supervised learning

In the supervised learning scenario, we work with a set of *objects* B, a set X called the *feature space* and a set Y called the *output space*. There is a mapping $f : B \mapsto X$ called *features extractor* that assigns *features* to objects.

There is a random variable $\mathcal{B}$ with domain B and another RV $\mathcal{Y}$ with domain Y and an unknown joint probabilistic distribution $\mathsf{P}(\mathcal{B}, \mathcal{Y})$.

We are also given a set of *samples* $T = \{t_1, t_2, \ldots, t_k\}$. Each sample t_i consists of an object $b_i \in B$ and a desired output $y_i \in Y$ such that (b_i, y_i) is sampled from the (unknown) joint probability distribution $\mathsf{P}(\mathcal{B}, \mathcal{Y})$. If the features extractor f is fixed, we can write $t_i = (x_i, y_i)$, where $x_i = f(b_i)$. We denote by $m : B \mapsto Y$ the (unknown) *optimal prediction model* where $\forall b \in B : m(b) = \mathsf{E}[\mathcal{Y}|\mathcal{B} = b]$.

The goal is to produce a *model* $\hat{m} : B \mapsto Y$ such that $\forall b \in B : \hat{m}(b)$ is close to $\mathsf{E}[\mathcal{Y}|\mathcal{B} = b]$. When the features extractor is fixed, we can write $\hat{m}(x_i)$ instead of $\hat{m}(b_i)$ where $x_i = f(b_i)$.

The model $\hat{m}$ is typically determined by a set of *parameters* - like parameters of linear regression, weights and biases of a neural net, etc. Lets denote the vector of parameters by θ and the parametrized model by $\hat{m}_\theta$. We will still use $\hat{m}$ when the parameter vector is not important or clear from context. See [39] section 2.4 for more exact definitions.

Example 1.10

> In the face recognition task, the set of objects B is a set of images of peoples' faces, typically in a *jpeg* or other format. Each image is normalized to some predefined size and then transformed to a bitmap format, i.e. a matrix of gray-scale of each pixel, or three matrices (i.e. a tensor) in case of RGB images. This encoding is done by the features extractor f. The output space Y is a set of IDs of people that we wish to recognize.

Methods of supervised learning are used for two fundamentally different tasks: *classification* and *regression*. In a classification task, Y is discrete and finite and there is no order defined on it (e.g. a set of colors, set of kinds of animals like $\{cat, dog\}$, etc.). In a regression task, Y is continuous and there is a non-trivial metric defined on Y. In this thesis, we only work with regression.

We use the following terminology. Given a sample $t_i = (b_i, y_i)$, a features extractor f and a predictive model $\hat{m}$,

- $x_i = f(b_i)$ we call the *features* of t_i
- y_i we call the *target* or *label* of t_i
- $\hat{y}_i = \hat{m}(x_i)$ we call the *output* on t_i.

1.2.3 Error function

Quality of the estimation is measured by so called *loss function* or *error function* $Err : Y \times Y \mapsto \mathbb{R}_0^+$. The error function quantifies the divergence between m and $\hat{m}$ on the set of samples. Given a sample $t_i = (x_i, y_i)$ and an output of the model $\hat{y}_i = \hat{m}_\theta(x_i)$, error of the model on this sample is $Err(y_i, \hat{y}_i)$. Given a set of samples, the error is calculated for each individual sample and the results are aggregated via a sum or an average. Given the set of training data T and the model $\hat{m}$, by $Err(T, \hat{m})$ we denote the cumulative error over all samples, i.e. $Err(T, \hat{m}) = \frac{\sum_{(x_i,y_i)\in T} Err(y_i, \hat{m}(x_i))}{|T|}$.

Machine learning algorithms in general try to minimize the cumulative error of $\hat{m}_\theta$ on the set of samples by successively adjusting the parameter vector θ.

In the regression setting, X and Y are tensor spaces and typical error functions are *Mean absolute error* and *Mean squared error*. We define those over vector spaces, for tensors the definitions are similar.

Definition 11 (Common error functions)**.** *Given $y, \hat{y} \in \mathbb{R}^d$, the **mean absolute error** (MAE) is calculated as*

$$Err_{MAE}(y, \hat{y}) = MAE(y, \hat{y}) = \frac{\sum_{i=1}^{d} |y[i] - \hat{y}[i]|}{d}$$

***Mean squared error** (MSE) is calculated as*

$$Err_{MSE}(y, \hat{y}) = MSE(y, \hat{y}) = \frac{\sum_{i=1}^{d} (y[i] - \hat{y}[i])^2}{d}$$

Distribution of error

We will talk in a greater detail about distribution of error. We will refer to this in section 5.5 where we describe benefits of the loss function we've designed.

For any loss function Err it holds that $y_i = \hat{y}_i \Rightarrow Err(y_i, \hat{y}_i) = 0$. In practice, it never happens that the cumulative error would reach zero. The model might not be able to represent the unknown function perfectly, or there could be some noise present in the training data, etc., hence the error will always be greater than 0.

For given training data $T = \{(x_i, y_i)\}_i^n$, model m_θ and a loss function Err, let's denote by Θ the set of all possible values of θ and by L the minimal value of error that is achievable on the given data. I.e.

$$L = \min_{\hat{\theta} \in \Theta} \left(\frac{1}{n} \sum_{i=1}^{n} \left[Err(y_i, \hat{y}_i^{\hat{\theta}}) \right] \right)$$

where $\hat{y}_i^{\hat{\theta}} = m_{\hat{\theta}}(x_i)$ and n is the number of training samples.

Different values of θ lead to different distribution of error among the samples. E.g. for some θ_1, error on the first sample will be smaller than the error on the second one: $Err(y_1, \hat{y}_1^{\theta_1}) < Err(y_2, \hat{y}_2^{\theta_1})$, while for some other value θ_2 it will be the other way around: $Err(y_1, \hat{y}_1^{\theta_2}) > Err(y_2, \hat{y}_2^{\theta_2})$. By adjusting θ, it is possible to reduce the error on individual samples. Once the cumulative error reaches the limit L, however, reducing the error on one sample is only possible at the cost of increasing the error on some other samples.

Loss functions differ in the way they distribute the error. E.g. *MSE* prefers values of θ that lead to even distribution of error among all samples. It prefers large number of small errors over fewer number of larger errors as can be seen in the following example.

Example 1.11

Let's compare behaviour of two loss functions: *MSE* and *MAE*. The set of training samples looks as follows $T = \{(2,2), (3,4), (4,3), (5,2), (6,3)\}$ and there is a model m_θ. Table 1.1 shows behaviour of the model with two different values of θ denoted θ_1 and θ_2.

x_i	y_i	$\hat{y}_i^{\theta_1}$	$\hat{y}_i^{\theta_2}$	$\lvert y_i - \hat{y}_i^{\theta_1} \rvert$	$\lvert y_i - \hat{y}_i^{\theta_2} \rvert$	$(y_i - \hat{y}_i^{\theta_1})^2$	$(y_i - \hat{y}_i^{\theta_2})^2$
2	2	3	4	1	2	1	4
3	4	3	4	1	0	1	0
4	3	3	4	0	1	0	1
5	2	3	4	1	2	1	4
6	7	3	4	4	3	16	9

Table 1.1: Distribution of error for two different loss functions.

MAE is calculated as $MAE = \frac{1}{n} \sum |y_i - \hat{y}_i|$. In our example, $MAE(T, m_{\theta_1}) = {}^7\!/_5$ and $MAE(T, m_{\theta_2}) = {}^8\!/_5$ so the value θ_1 is preferred. Using MSE, we get $MSE(T, m_{\theta_1}) = {}^{19}\!/_5$ and $MSE(T, m_{\theta_2}) = {}^{18}\!/_5$ so using MSE, θ_2 would be preferred.

1.2.4 Machine learning models

There are many types of ML models that differ in the way they work and especially in the type of features they require. We will distinguish two categories of models: *one-to-one* and *sequence-to-sequence*. *One-to-one* models work with feature vectors of **fixed size** and produce a **fixed size** vector as their output. Size and shape of inputs and outputs are hard-coded in the model hence it only accepts inputs of certain type. This is the case for most *standard* models like linear regression, decision trees, support vector machines, feed-forward neural networks and others whose input space is $\mathbb{R}^d$ for some fixed dimension d, or in general a tensor space $\mathbb{R}^{d_1} \times R^{d_2} \times \cdots \times R^{d_k}$ for some fixed $d_1, d_2, \ldots d_k \in \mathbb{N}$.

Sequence-to-sequence models on the other hand allow variable size inputs to be given to the model and size of their output varies as well. These are often used for processing time series data or texts. Among the most popular models in this category we can mention *Long short-term memory (LSTM)* or *Gated recurrent unit (GRU)*. See [35] for details.

Some models like linear regression or symbolic regression make strong assumptions about structure of the problem and hence they can work even with a very small number of training samples.

More complex models like shallow or deep neural nets can better fit the data but require larger amount of training data to work properly and avoid overfitting. See [35, 39] for details.

In this thesis we only work with *one-to-one* models, specifically with deep feed-forward neural networks.

1.3 Heuristic Learning

Heuristic learning (HL) is an application of supervised ML to estimate goal distances of states of some planning problem. The set of objects B in this case is a set of states of planning problems, output space is $\mathbb{R}_0^+$. Training samples are pairs $(s_i^P, h^*(s_i^P))$ where s_i^P is a state and $h^*(s_i^P)$ is its goal-distance in P. The system involves a ML model that is trained to approximate the mapping $s \mapsto h^*(s)$. The trained model is then used as a heuristic function during search.

Example 1.12

> The HL approach can be used to learn an efficient combination of several heuristics. Given a set of heuristics $H = \{h_1, h_2, \ldots, h_k\}$ and a set of training samples $(s_i^P, h^*(s_i^P))$, one can use linear regression to find values of parameters $\{c_0, c_1, \ldots, c_k\}$ such that $c_0 + \sum_{j=1}^{k} c_j \cdot h_j(s)$ is close to $h^*(s)$ on the set of samples.
>
> This way of combining the heuristics might give better result than using simple maximum or sum.

1.3.1 Variants of HL

HL can be used in several different scenarios based on the generalization capabilities that are required of the model. No unified categorization of HL variants currently exists in the literature so we propose the following three categories.

Type I HL

By *Type I* we denote the variant of HL that aims at constructing a heuristic tailored for a single problem. In this case, the model will only generalize over different states of the same problem, i.e. the set of predicates, predicate symbols and constants as well as the goal condition will always be the same.

Given a problem P and a set of training data $(s_i^P, h^*(s_i^P))$, the goal is to construct a heuristic $h : S^P \mapsto \mathbb{R}$ and then use it to solve the problem P as fast as possible. Typically, the time required to construct the heuristic is considered part of the solving process hence we want to minimize the sum of *heuristic-construction time* and *search time*.

This variant is sometimes referred to as *per-instance heuristic learning*, e.g. in [23].

Type II HL

In the *Type II* scenario we aim at constructing a heuristic applicable to a whole domain, i.e. we generalize over problems from the same domain. In this case, the model needs to be able to handle inputs with variable set of constants and variable goal conditions. The set of predicate symbols as well as action-templates will remain the same in all inputs.

We are given several problems $\mathcal{P} = \{P_1, P_2, \dots P_k\}$ from the same domain $\mathcal{W}$ and a set of training data $\{(s_i^{P_j}, h^*(s_i^{P_j})) \mid P_j \in \mathcal{P}\}$, the task is to construct a heuristic h applicable to any problem from $\mathcal{W}$. (I.e. not only to problems in $\mathcal{P}$.)

Typically, the time required for training the model is **not** considered a part of the solving process. It is considered a pre-processing, or a domain analysis phase that could take several hours or even days. After the model is trained, new problems from the same domain can be solved quickly.

Unlike the *Type I*, here the ML model needs to be able to handle problems of different sizes and with different goal conditions hence the features extractor must be much more sophisticated.

We could further distinguish systems that generalize over *fluent* predicates only (see section 1.1.1), e.g. over problems on a fixed "route map" and systems that generalize over both *fluent* and *rigid* predicates. There is also a significant difference between working with *action costs* vs. *unit costs* as described in the following section.

In [23], authors refer to *Type II HL* as to *per-domain heuristic learning* without explicitly distinguishing the *fluents*-only vs. *full* nor the action costs variants.

Type III HL

Type III HL deals with the task of automatically creating a domain-independent heuristic. The setting is similar as in the previous case except that the training problems come from several different domains and the heuristic have to be applicable to previously unseen domains.

This requires a very flexible features extractor and a ML model with huge expressive power like *Recurrent NN*, *Neural Turing machine* or models based on *Genetic programming*.

In this thesis, we deal with the Type II scenario.

1.3.2 Action costs

Generalization capabilities of the model are also related to costs of actions. In the *Type I* scenario, there are no difficulties. Any model is applicable to both *unit-cost* and *non unit-cost* problems in the same way.

In the case of *Type II* HL, domains with unit-costs are easier as they don't require the model to generalize over costs of actions. Domains where action costs are present can be further divided into two categories:

1. Costs are *domain-dependent*: cost of action only depends on *name* of the action. (Denoted *DDCosts* for future reference.)

2. Costs are *instance-dependent*: cost of action depends on both name of the action and its arguments. (Denoted *IDCosts*.)

In the PDDL, we are given *templates* of actions that define *name*, *number of arguments*, *types of arguments* and preconditions and effects described relatively, i.e., in terms of arguments. These templates are referred to as *operators*. When we specify arguments of the operator, we get an instance which is referred to as *instantiated action*.

Example 1.13

> In Zenotravel, the operator *Fly* has five arguments:
> *Fly(?a - Aircraft, ?c1, ?c2 - City, ?fl_before, ?fl_after - FuelLevel)*
> Instantiated action can then look as follows:
> *Fly(plane1, city1, city2, fl3, fl2).*

With *DDCosts*, costs of all instances of the same operator are the same while with *IDCosts*, the costs of instances could differ.

Example 1.14

> *Zenotravel* features actions *board*, *debark*, *fly* and *refuel*. Originally, costs of all actions are 1. If we changed cost of action *fly* to 10 and *refuel* to 5, for example, the domain would still have *DDCosts*.
>
> If the cost of *fly* actions depended also on cities between which the plane flies, like *c(fly(city1, city2))* = 10, *c(fly(city1, city3))* = 18, etc, we would get *IDCosts*.

With *DDCosts*, costs of actions are the same across the whole domain, i.e., they need not to be encoded into features of states. When predicting $h^*(s)$, the model needs to estimate *the number of applications of each operator* required to reach the goal. With unit costs, the model only needs to predict the *sum* of number of applications of each operator which is always easier. Besides this, however, the *DDCosts* variant is still quite similar to the unit-costs scenario.

The scenario with *IDCosts* is much more complex. Not only must costs of all instantiated actions be encoded in the features vector but the model needs to be able to generalize over problems with arbitrary action costs.

Just by changing the action costs, it is already possible to encode NP-hard problems, such as *TSP*. Imagine the following scenario: We are given a fixed set of cities and a single operator $move(?c1, ?c2)$ to travel between the cities. Cost of each instantiated action corresponds to distance between the respective cities. The goal is to visit all cities and minimize plan cost. The HL system for this domain would have to be able to learn a general algorithm for solving TSPs.

1.3.3 Usage scenarios of HL

The Type II HL requires significant amount of time to train the model hence it might seem that the approach is not practically applicable. The time demands do limit usage of this method but in many scenarios it can prove beneficial.

When there is a need to solve a business problem using action planning, the AI practitioner has several options:

- use a domain-independent solver
- use existing domain-specific solver
- develop a new domain-specific solver
- use heuristic learning

A domain-independent solver can be used of-the-shelf as long as there is a PDDL model of the problem. This is the easiest and the most common way. Efficiency of domain-independent solvers however varies and on some domains they perform poorly.

Domain-specific solvers perform much better but they are only applicable to a narrow range of domains. There are planning and optimization problems like *Traveling Salesman Problem* or *Vehicle Routing Problem* for which appropriate solvers exist. In practice, however, the problems are very specific. They often involve additional constrains and parameters like types of vehicles, number of drivers, their working hours etc. It is unlikely that solver would already exist for such specific problem. Dropping the additional constrains would lead to finding inferior or even infeasible solutions.

Another option is to develop a solver specifically for the problem at hand as has been done in [94] for example. Such endeavour requires cooperation between AI researchers, SW developers and domain experts and typically takes a lot of time and resources.

Heuristic learning allows us to combine advantages of the other approaches. It constructs a strong heuristic which provides good performance and doesn't require access to domain knowledge nor time of AI researchers and developers, i.e. it can be used of-the-shelf. This could be very useful in business applications. Successes of AlphaZero and other learning engines indicate that the ML approach could even achieve better performance than human designed systems. It might therefore be advantageous to use HL even if the domain expert was available.

From the ML perspective, heuristic learning has another significant advantage: training data can be labeled automatically. This saves us the time and resources required for humans to label the samples which is required in most ML applications.

The biggest drawbacks of the method are time required to train the model and the need for training problems. Due to these limitations, the technique is only applicable in specific situations.

In practice, the ideal use case scenario is as follows.

- a company needs to solve planning tasks on a daily basis
- tasks are similar to each other
- when new task arrives, it must be solved quickly
- there is a large number of historical tasks available

The situation resembles operations of a shipping company that needs to plan routes for deliveries every day, or any other company whose operations involve navigation, for example. In such scenarios, it is acceptable for the company to spend several days or even weeks training the model as long as it will be cheap and the resulting heuristic will be efficient.

Search algorithms are typically tested in a different setting: a problem description is given and the clock starts ticking immediately. The goal is to solve the problem as fast as possible while quality of the solution is taken into account in the evaluation. See *https://ipc2018-classical.bitbucket.io/#tracks* for example. In that setting, our technique is not applicable. We argue, however, that from a practical point of view, such setting is very rare since in practice we always have historical data or some other source of information that we can use.

In the past, the IPC also featured a *learning track*[4] in which several small problems are given as inputs together with their optimal solutions and some time is allowed to analyze the data. After the analysis, a set of new problems similar to the previous ones is given and the task is to solve each of them quickly. This is much closer to our intended use-case.

1.4 Related Works

Many attempts have been made to utilize ML in planning and in general search. See [51] and [95] for surveys. ML has been used to learn reactive policies ([36, 61]), control knowledge ([97]), for plan recognition ([5]) and for other planning-related tasks ([58]). ML tools are also often used to combine several heuristics ([76, 27]) and in particular to help portfolio-based planners to efficiently combine multiple search algorithms ([11, 81]). [92] presents the *Explicit Estimation Search* algorithm that learns error of the heuristic online during search and adjusts future estimates accordingly.

A lot of papers exist that utilize ML in neoclassical planning paradigm (partial observability, non-deterministic actions, extended goals etc.) mostly to learn reactive policies ([91, 29]). Several attempts also exist to utilize reinforcement learning for planning like [37, 73, 30]. These techniques are not directly related to our work.

[4] https://www.cs.colostate.edu/ ipc2014/

1.4.1 Heuristic learning

Heuristic learning was investigated by [1] where the authors used a bootstrapping procedure with a NN to successively learn stronger heuristics using a set of small planning problems for training. The paper proposed an efficient way of generating training data based on switching between learning and search phases. The technique become popular and was successfully used by other authors ([13, 93, 24]). We use a modified version of this technique as well. A domain-independent generalization of this approach was published later in [31].

Type I HL

Most papers deal with the *Type I HL* scenario and they use either a set of simple heuristics like PDBs as features ([2, 8]) or SAS^+ encoding as features ([93, 23, 24]). In [34], authors combined heuristic values with other information about the problem like the number of objects. Serious attempt to use other kind of features was made in [97].

[23] provides a comprehensive study of hyper-parameters for *Type I HL*. Authors experiment with different input encodings (SAS^+ vs. one-hot), classification vs. regression, different ML models and NN architectures and they investigate the amount of training data needed to achieve competitive results. The follow-up paper ([24]) adds a comparison of different training data-generation methods and a comparison of efficiency between *Type I* and *Type II HL* systems.

In [67] authors propose yet another method for sampling state-space of the problem which leads to a better distribution of training data and a better performance of the learned heuristic. A comprehensive analysis of sampling methods for HL is available in [4].

Type II HL

Several *Type II HL* systems also exist [80, 52]. The method proposed in [80] utilize recurrent NN with a new elaborate way of extracting features based on hyper-graphs. Authors report competitive results against the LM-cut heuristic. The technique is able to address *Type III HL* as well which is also investigated in the paper, so far with limited success.

In [52] authors use a feed-forward network and they encode description of the state as well as goal conditions in the features vector. They propose a *canonical abstraction* to keep dimensions of the features vector fixed regardless of the number of objects in the problem. The paper reports competitive results against hand-coded heuristics.

Most papers use a standard *MSE* loss function without any adjustments and the effects of choice of the loss function is not studied. Few exceptions exist: in [93] the authors proposed a modification to the loss function used during the training to bias the model towards under-estimation which increased quality of solutions found during the subsequent search.

1.4.2 Performance guarantees

A lot of effort has been devoted to provide theoretical performance guarantees for NNs and other ML models, i.e. bounds on the prediction error under various conditions. See [35], section 6.4. In planning, the situation is similar. Many heuristic search algorithms provide optimality guarantees or at least some bounds on solution quality. See [87, 33] for example.

There is very little work, however, on providing any theoretical guarantees in situations where planning is combined with learning or with data-driven techniques in general. The only related work is *Probably bounded sub-optimal search* framework introduced in [85] and further studied in [86, 88]. Authors use collected samples in a form of $(h(s_i), h^*(s_i))_i$ to estimate heuristic accuracy and provide probabilistic bounds on solution quality. Besides this, utilizing data is still very uncommon in combinatorial search.

2. Heuristics in an ML Age

In this chapter, we review the role of heuristics in forward search algorithms. We mention several well known theorems about performance of A^* algorithm and we analyze properties of heuristics from the machine learning perspective.

We also present the notion of heuristic adjustment - an automated data-based modification of existing heuristics that we've developed.

2.1 Heuristics and Search Algorithms

Current state-of-the-art forward search algorithms use heuristics to guide the search towards promising areas of the state-space. The heuristic imposes an order on the set of open nodes which determines what nodes will be expanded first. Properties of the heuristic have significant impact on performance of the search algorithm - both run-time and solution quality. The most important performance guarantees are presented as theorems 1 and 2.

Definition 12 (Cost of A* solution)**.** *Let P be a planning problem and h a heuristic. For a state $s \in S^P$, we denote by $A_h(s)$ the **cost of plan** that A^* algorithm finds from s to some goal state of P using h as heuristic. If no such plan exists, we set $A_h(s) = \infty$.*

Definition 13 (Admissible heuristic)**.** *Given a planning problem P, heuristic h is called **admissible on P** if $\forall s^P \in S^P : h(s^P) \leq h^*(s^P)$. The heuristic is called **admissible** if it is admissible on all problems to which it can be applied.*

Theorem 1 (Optimality of A^*)**.** *Let P be a planning problem and h an admissible heuristic then $A_h(s_0^P) = h^*(s_0^P)$, i.e., the algorithm finds optimal solution.*

Proof. See Theorem 2.10. in [20]. □

Definition 14 (ϵ-admissible heuristic)**.** *Given a planning problem P and $\epsilon \geq 1$, heuristic h is called **ϵ-admissible on P** if $\forall s^P \in S^P : h(s^P) \leq \epsilon \cdot h^*(s^P)$. The heuristic is called **ϵ-admissible** if it is ϵ-admissible on all problems to which it can be applied.*

Theorem 2 (Bounded suboptimality of A^*)**.** *Let P be a planning problem and h an ϵ-admissible heuristic then $A_h(s_0^P) \leq \epsilon \cdot h^*(s_0^P)$, i.e., cost of the solution is no more than ϵ-times greater than cost of the optimal solution.*

Proof. See Lemma 6.2. in [20]. □

Another important property of a heuristic is its informedness.

Definition 15 (Heuristic domination)**.** *Let h_1, h_2 be heuristics. We say that h_2 **dominates** h_1 if $\forall s : h_1(s) \leq h_2(s) \leq h^*(s)$.*

Definition 16 (Set of expanded states)**.** *Let h be a heuristic and s^P a state. By $A_h^S(s)$ we denote a **set of states that A^* expanded** when searching from s and using h as heuristic.*

Theorem 3 (Space-efficiency of A^*)**.** *Let P be a planning problem and h_1, h_2 heuristics such that h_2 dominates h_1. Then $A^S_{h_2}(s^P_0) \subseteq A^S_{h_1}(s^P_0)$.*

Heuristics that produce higher estimates lead to finding solutions quickly, even when they're not admissible. Lower estimates on the other hand lead to finding solutions of higher quality. There is a tradeoff between speed and solution quality.

2.2 The Process of Heuristic Estimation

Role of the heuristic is to estimate $h^*(\cdot)$ which is not known and cannot typically be computed during the search due to the high computational complexity of such task. Time complexity of A^* on a planning problem depends on accuracy of the heuristic used. Theorem 3 in [68] (page 186) states that if $\forall s \in S^P : h(s) = h^*(s)$ then A^* with h finds optimal solution to P after expanding only polynomial number of nodes. Given the fact that *NP-hard* planning problems do exist and assuming that $P \neq NP$, it is not possible for a polynomial-time computable h to be equal to h^* on all states. We rely here on an intuitive notion of complexity. For more exact definitions of complexity of planning domains see [42]. For a review of several more recent complexity results see [45].

The notion of *heuristic residue* (also called *error of heuristic*) captures the difference between $h^*(s)$ and $h(s)$.

Definition 17 (Heuristic residue)**.** *Let h be a heuristic. For a given state s^P, **residue** of h on s^P denoted $R_h(s^P)$ is $R_h(s^P) = h^*(s^P) - h(s^P)$.*

The process of calculating the heuristic estimate can be divided to two phases. Given the set of states S, we first divide it into categories and then assign an estimate to each category. There is a set of categories Δ, a **dividing** component $d : S \mapsto \Delta$ and an **estimating** component $e : \Delta \mapsto \mathbb{R}$.

Every existing heuristic h can be disassembled into these two components d_h, e_h such that $\forall s \in S : h(s) = e_h(d_h(s))$ and $|\Delta| << |S|$.

This view on heuristics is very useful for our work as heuristics constructed by HL systems work in the similar way. They extract *features* of states using the features extractor $f : S \mapsto \mathbb{R}^d$ and then use an ML model $M : \mathbb{R}^d \mapsto \mathbb{R}$ to produce the estimate based on the features. Δ in this case corresponds to the "features-space" of the heuristic.

We will show how operations of any heuristic can be viewed in this way. When computing $h(s)$, the heuristic first creates a simplified inner representation $F(s)$ of the state s and then compute $h(s)$ as a function of $F(s)$. If the heuristic is represented by an algorithm, $F(s)$ can for example be defined as a sequence of values of local variables of that algorithm during the computation.

The inner representation that the heuristic uses is implicit, hence not directly observable but its approximation can be reconstructed by analyzing behavior of the heuristic.

Definition 18 (Heuristic equivalence)**.** *Let h be a heuristic and P a problem. A **heuristic equivalence** $\sim_h$ is an equivalence relation on S^P defined as*
$s_1 \sim_h s_2 \Leftrightarrow h(s_1) = h(s_2)$.

Given a heuristic h, we can define the set of categories Δ as a set of equivalence classes of $\sim_h$. We will call this set the **Δ-sets of h**.

Definition 19 (Δ-sets)**.** *Let h be a heuristic, P be a problem and $s \in S^P$ its state. By Δ_s^h we denote the set of states with the same heuristic estimate as s, i.e. $\Delta_s^h = \{s_i \in S^P \mid s_i \sim_h s\}$.*

We will just write Δ_s when the heuristic is clear from context. We will also use $h(\Delta_s)$ to denote value of the heuristic on states in Δ_s. The value is the same for all $s \in \Delta_s$.

2.2.1 Statistical perspective

For given s, we can rewrite the formula for residue as $h^*(s) = h(s) + R_h(s) = e_h(d_h(s)) + R_h(s) = e_h(\Delta_s) + R_h(s)$. In this equation, $h^*(s)$ is the value we want to estimate, $e_h(\Delta_s)$ is the part of the information that h can compute and R_h is the residue that we cannot compute using only h.

For given heuristic h and state s let's define

- $R_h^-(s) = \min\{R_h(s_i) \mid s_i \in \Delta_s\}$
- $R_h^+(s) = \max\{R_h(s_i) \mid s_i \in \Delta_s\}$

Then $\forall s : R_h(s) \in [R_h^-(s), R_h^+(s)]$. From this point of view, $R_h(s)$ can be seen as a random noise with an unknown distribution over interval $[R_h^-(s), R_h^+(s)]$. Values $R_h^-(s)$ and $R_h^+(s)$ are the same for all states $s \in \Delta_s$ hence we can write $R_h^-(\Delta_s)$ instead of $R_h^-(s)$ and similarly for R_h^+.

Example 2.1

Consider the problem whose state space is depicted in figure 1.2. Lets assume we have two admissible heuristics h_1 and h_2 that give us estimates as shown in table 2.1.

state s	$h^*(s)$	$h_1(s)$	$R_{h_1}(s)$	$h_2(s)$	$R_{h_2}(s)$
s_1	3	2	1	1	2
s_2	2	2	0	1	1
s_3	0	0	0	0	0
s_4	4	2	2	1	3
s_5	1	1	0	0	1
s_6	0	0	0	0	0

Table 2.1: Example of estimates and residues of two heuristics over a simple state-space.

Lets analyze the Δ-sets of the two heuristics. The relation $\sim_{h_1}$ has three equivalence classes. Table 2.2 shows them together with their corresponding heuristic values and bounds R^- and R^+. Table 2.3 shows the same for heuristic h_2.

set $\Delta_s^{h_1}$	$h_1(\Delta_s)$	$R_{h_1}^-(\Delta_s)$	$R_{h_1}^+(\Delta_s)$
$\{s_3, s_6\}$	0	0	0
$\{s_5\}$	1	0	0
$\{s_1, s_2, s_4\}$	2	0	2

Table 2.2: Δ-sets of heuristic h_1.

set $\Delta_s^{h_2}$	$h_2(\Delta_s)$	$R^-_{h_2}(\Delta_s)$	$R^+_{h_2}(\Delta_s)$
$\{s_3, s_5, s_6\}$	0	0	1
$\{s_1, s_2, s_4\}$	1	1	3

Table 2.3: Δ-sets of heuristic h_2.

Values R^- and R^+ delimit the interval in which the unknown residue lies. E.g. if $h_1(s)$ gives us an estimate of 2, the actual value $h^*(s)$ equals to $2 + x$ where x is unknown and lies in $[0, 2]$.

As we saw, computing a heuristic can be viewed as estimating a noisy random variable. The issue of value estimation is well studied in the field of statistics that deals estimating mean, variance or other properties of unknown random variables. ML models are based on statistical approaches as well hence it is useful to compare heuristic distance estimators to these techniques.

There are three basic types of estimates:

1. point estimate
2. interval estimate
3. estimation of distribution

A point estimate is a single number that is in some sense *close* to the unknown value. An interval estimate provides a confidence interval that contains the unknown value with high probability and estimation of the distribution furthermore indicates how likely it is that the estimated value lies in some specific part of the interval.

Example 2.2

Given a sampling $x_1, x_2, \ldots x_k$ of some unknown RV $\mathcal{X}$, we may use average of $x_1, x_2, \ldots x_k$ as a point estimate of $\mathsf{E}(\mathcal{X})$.
If in addition we knew that $\mathcal{X} \sim N(\mu, \sigma^2)$ for some unknown μ and σ^2, we could use the sampling to construct and interval $[a, b]$ such that $\mathsf{P}[\mathsf{E}(\mathcal{X}) \in [a, b]] \geq 0.95$, for example.
We could also come up with a function $\hat{d}_{\mathcal{X}}$ such that $\forall c : \mathsf{P}[\mathcal{X} < c] \geq \hat{d}_{\mathcal{X}}(c)$ which would give us an estimation of distribution of $\mathcal{X}$.

If we compare heuristics to those techniques, we see that non-admissible heuristics work similarly to point estimates as they produce a single value $h(s)$ such that $|h^*(s) - h(s)|$ is as small as possible. $h(s)$ can be either smaller of greater than $h^*(s)$.

Admissible heuristics produce a single value as well but in this case the result should be interpreted as an interval estimate. If the heuristic is admissible then $h(s) \leq h^*(s)$ so the heuristic actually provides an interval $I = [h(s), \infty)$ and guarantees that $h^*(s) \in I$. This can be seen as a 100%-confidence interval. No upper bound nor an estimate of distribution is provided, i.e. the heuristic doesn't tell us how likely it is that h^* lies in some specific part of the interval.

2.3 Components of the Heuristic

It is possible to analyze the two components (d_h and e_h) of a heuristic separately. There are strong connections between properties of the HL system and behavior of the two components of the resulting learned heuristic. We will first analyze the desired behavior of these components for the learned heuristic and then point out the correspondence between settings of the HL system and properties of the learned heuristic.

2.3.1 Desired behavior

Ideally, the d-component should divide states to categories with the same h^*, i.e., for an optimal d-component it should hold that

$$\forall s_1, s_2 : d(s_1) = d(s_2) \Leftrightarrow h^*(s_1) = h^*(s_2).$$

For an $NP - hard$ planning problem such function d cannot be computed in polynomial time unless $P = NP$. (See the following example.) This shows that the dividing component is actually the hard and interesting part of any heuristic estimation.

Example 2.3

Let's assume we have a black-box for computing the optimal d-component. We can use it to determine for any given states s_1, s_2 whether $d(s_1) = d(s_2)$, i.e. whether $h^*(s_1) = h^*(s_2)$.

Now, given a nontrivial Sokoban[a] problem with the initial state s_0, we can easily construct a state s_x of the same problem that is unsolvable. This can be achieved by adding an *unrecoverable configuration* (see [14]), e.g. placing a crate to a corner. Now we ask the black-box whether $h^*(s_0) \stackrel{?}{=} h^*(s_x)$. They will have the same value if and only if the state s_0 is unsolvable.

We can use the black-box to determine whether any given Sokoban instance is solvable. The decision-problem version of Sokoban, however is known to be PSPACE-hard. See Theorem 2.1 in [17].

[a] https://en.wikipedia.org/wiki/Sokoban

In ML applications, the features-space has a form of vector space over $\mathbb{R}$ on which we have a natural metric: Euclidean distance. The ML system works best if there is a strong correlation between $dist(f(s_1), f(s_2))$ and $|h^*(s_1) - h^*(s_2)|$, $\forall s_1, s_2 \in S$, i.e., if states whose targets are close to each other have similar features.

As for the e-component, it is not clear what the optimal behaviour is. It affects both admissibility and informedness of the heuristic. For a given d-component, we need to assign an estimate to each set $\Delta_s = \{s_i \in S \mid d(s_i) = d(s)\}$. Let's denote $\min^s = \min_{s_i \in \Delta_s}(h^*(s_i))$ and $\max^s = \max_{s_i \in \Delta_s}(h^*(s_i))$.

To guarantee admissibility, we need $e(\Delta_s) \leq \min^s$. It's never useful to produce an estimate $e(\Delta_s) < \min^s$ hence the most informed admissible heuristic (with the given d-component) should produce $e(\Delta_s) = \min^s$. If admissibility is not required, a different value from the interval $[\min^s, \max^s]$ might be preferred.

Knowing the distribution of $h^*(s_i)$ over $s_i \in \Delta_s$, we can take quantiles of the distribution into account. E.g., we may produce an estimate $e(\Delta_s) = \beta$ such that $\beta \leq h^*(s_i)$ for at least 80 % of states $s_i \in \Delta_s$, i.e. guaranteeing that the heuristic is admissible on at least 80 % of states, etc. The final choice depends on user's preferences with respect to the speed vs. solution quality trade-off.

2.3.2 HL connections

In the HL system, the d-component of the resulting heuristic is fully determined by the choice of features extractor. The training data or even the choice of model (neural network, linear regression, etc.) have no impact on it. I.e., if two states s_1 and s_2 have the same features, the learned heuristic will always produce the same estimate for both of them.

The e-component can be controlled by the choice of loss function. When using MSE, the model will prefer $e(\Delta_s) = \beta$ such that $\sum_{s_i \in \Delta_s}(h^*(s_i) - \beta)^2$ is minimal. Other loss functions might lead to producing an average of $\{h^*(s_i) \mid s_i \in \Delta_s\}$, a median or some other value.

The loss function is not evaluated over the whole S but only over the given set of training samples hence the choice of training data and other techniques like regularization also affect the e-component. Furthermore, ML models are able to interpolate the features space, i.e. when calculating $e(\Delta_s)$, they take into account also data points in other categories than just Δ_s, i.e., those that have similar features.

These circumstances cannot be directly controlled but by a proper choice of the loss function, the user can project their preferences about the e-component of the learned heuristic.

2.4 Heuristic Adjustments

The presented component-based view can help tune hyper-parameters of a HL system and can also be used to analyze existing heuristics. Given a heuristic h and a set of training data in a form $(s_i, h^*(s_i))$, we can construct an approximation of Δ-sets of the heuristic as well as compute bounds on residues $R^-(\Delta_s)$ and $R^+(\Delta_s)$ as in example 2.1.

We can then use this information to discover possible discrepancies in behavior of the heuristic and even fix them automatically. We call this process **heuristic adjustment**. By adjusting a heuristic h, we actually create a new heuristic h' whose d-component is the same as the one of h and the e-component is replaced by one computed from the training samples.

Example 2.4

In table 2.3 we see that $R^-_{h_2}$ is greater than 0 for the second Δ-set. This is undesirable as it means that the heuristic systematically underestimates the corresponding set of states. Simply by increasing the estimate on these states, we can come up with a heuristic h'_2 that is more informed than h_2 and is still admissible. The adjusted heuristic in this case looks as follows:

$$h'_2(s) = \begin{cases} h_2(s) + 1 & if\ h_2(s) = 1 \\ h_2(s) & otherwise \end{cases}$$

The new heuristic can be evaluated in the same time as h_2 (up to a small additive constant) and dominates the original. Using h'_2 will always yield the same or better results than using h_2 with respect to both search-speed and solution quality according to theorems 1 and 3.

We propose three types of adjustments: *Shift-adjustment*, *Avg-adjustment* and *Min-adjustment*.

2.4.1 Shift-adjustment

As example 2.4 shows, the lower bound R_h^- should never be greater than 0. The *shift-adjustment* is based on identifying such discrepancies and fixing them by shifting the estimate appropriately.

Underestimation such as the one in the example should never occur with state-of-the-art domain-specific heuristics as it harms the performance and is avoidable. Author of the heuristic should be able to identify and remove such anomaly.

With domain-independent heuristics, however, this can happen. The systematical underestimation could only occur on some specific domain and shifting the value globally could make the heuristic inadmissible on other domains. It is not possible for the author of heuristic to manually check its behaviour on all existing domains and adjust it to each such domain. With the ML approach, however, we can do exactly that as the adjustment is applied automatically and is domain-specific.

It is also possible to shift the heuristic the other way, i.e. identify Δ-sets on which a systematical overestimation occurs and mitigate it by lowering the value. We identify both systematical underestimation and overestimation. We denote the resulting heuristic h^{shift} and define it as follows.

$$h^{shift}(s) = \begin{cases} \min^s, & if\ h(s) < \min^s \\ \max^s, & if\ h(s) > \max^s \\ h(s) & otherwise \end{cases}$$

2.4.2 Avg-adjustment

Analyzing the distribution of the residue among states can also be used to adjust the heuristic. In particular, we would like to determine if there is a dependence between $h(s_i)$ and $R_h(s_i)$ for $s_i \in S^P$. If such dependence is present, the ML model could utilize it to predict $R_h(s)$ from $h(s)$.

Example 2.5

If the heuristic would always predict $h^*(s)/2$ (i.e. $\forall s : h(s) = h^*(s)/2$), then given enough samples $(h(s_i), h^*(s_i))$, the model would easily learn the dependence and would be able to predict $h^*(s)$ accurately.

This can be achieved by calculating average of $h^*(s)$ over each Δ-set and use that as the estimate. We call this procedure *avg-adjustment* and we denote the resulting heuristic h^{avg}. The heuristic is defined as follows:

$$\begin{aligned} h^{avg}(s) &= average\{h^*(s_i) \mid s_i \in \Delta_s\} \\ &= average\{h^*(s_i) \mid s_i \in S^P, h(s_i) = h(s)\} \end{aligned}$$

See the following example.

Example 2.6

We do this modification for heuristic h_2 defined in table 2.1. There are two Δ-sets Δ_{s_1} and Δ_{s_3}. Δ_{s_3} contains states s_3, s_5, s_6. When we average their h^*, we get $1/3$. When we do the same for Δ_{s_1}, we get the average of 3. Table 2.4 compares behaviour of h_2 and h_2^{avg}.

state	h^*	h_2	R_{h_2}	h_2^{avg}	$R_{h_2^{avg}}$
s_1	3	1	2	3	0
s_2	2	1	1	3	-1
s_3	0	0	0	$1/3$	$-1/3$
s_4	4	1	3	3	1
s_5	1	0	1	$1/3$	$2/3$
s_6	0	0	0	$1/3$	$-1/3$

Table 2.4: Comparison of heuristics h_2 and h_2^{avg}

We can see that h_2^{avg} has lower average error but we have lost admissibility of the heuristic as on states s_2, s_3 and s_6 h_2^{avg} overestimates the true distance.

2.4.3 Min-adjustment

By taking *minimum* of $h^*(\cdot)$ instead of average, we can make the heuristic admissible. We denote such heuristic h^{min} and define it as follows.

$$\begin{aligned} h^{min}(s) &= \min^s \\ &= \min\{h^*(s_i) \mid s_i \in S^P, h(s_i) = h(s)\} \end{aligned}$$

Example 2.7

The following table 2.5 compares behavior of heuristic h_2 defined in 2.1 and its min-adjustment h_2^{min}.

state	h^*	h_2	R_{h_2}	h_2^{min}	$R_{h_2^{min}}$
s_1	3	1	2	2	1
s_2	2	1	1	2	0
s_3	0	0	0	0	0
s_4	4	1	3	2	2
s_5	1	0	1	0	1
s_6	0	0	0	0	0

Table 2.5: Comparison of heuristics h_2 and h_2^{min}

2.4.4 Adjustments vs. weighting

There is a popular technique for increasing the heuristic value called *weighting*. Given a heuristic h and a weight $w \in R, w > 1$, a weighted heuristic is defined simply as $h^{(w)}(s) = w \cdot h(s)$. The weighted heuristic is more informed and hence might find solutions faster but typically it is not admissible so it doesn't guarantee optimality of the solution. If h is admissible then $h^{(w)}$ is w-admissible so theorem 2 can be used to obtain a bound on solution quality.

It is also possible to down-weight an inadmissible heuristic, i.e. to use $w < 1$. This does slow down the search but should improve the solution quality. By adjusting the weight, the user can choose the speed vs. quality ratio that suits their usage scenario.

The presented modifications - especially the *averaging* - serve the similar purpose as weighting, i.e. making the heuristic more informed. There is however a fundamental difference between weighting and using a data-driven approaches like *averaging* or *shifting*.

Weighting increases the heuristic value for all states evenly. There is no guarantee that the new value will be closer to h^* or that the weighted heuristic will perform any better during search than the original. There is also no guideline on how to choose the weight nor any theoretical reasoning why weighting is the correct way to improve the heuristic. There are many other transformations we could apply to the heuristic, like $h(s) + c$, $h(s) \cdot h(s)$, $h(s) \cdot \log(h(s))$, $h(s) + c \cdot (\log(h(s)) - 1)$, etc.

Heuristic adjustments are related to the work of Roni Stern at al. ([86, 88]) who also developed a way of utilizing data to modify heuristic behavior. Authors report significant improvements over weighted A^* and some other bounded-suboptimal algorithms.

2.4.5 Properties

The min-adjustment yields the most informed admissible heuristic with the given d-component. The avg-adjustment yields a heuristic with the lowest mean absolute error (MAE), given the d-component and the shift-adjustment corrects some obvious errors of h. h^{shift} will always have the same or lower MAE than h without compromising its admissibility nor informedness.

Unlike weighting, these data-driven modifications allow targeting a specific types of states and provide theoretical guarantees of the behaviour. They give the user a direct control over the distribution of $\mathsf{P}[h^*(s) = x \mid h(s) = y]$. E.g., it

is possible to modify the heuristic in such a way that $\forall s : h^{modified}(s) - h^*(s) < 2$ or that the number of states on which the heuristic overestimates is less than 5% of all states, etc.

We are actually only constructing **approximations** of heuristics h^{min}, h^{shift} and h^{avg} as we don't calculate the value over the whole S but over the training data only. The theoretical properties like admissibility can only be guaranteed on the set of training data. If the set of training samples is too small or poorly chosen then the heuristics might not exhibit the advertised behavior when deployed.

The properties can only be guaranteed "asymptotically", i.e., as the set of training samples converges to S, the number of states that violates the property goes to zero.

We've create adjusted variants of two domain-independent heuristics and compared their performance with the originals on a set of planning benchmarks. Results are provided in the *Experiments* section.

Heuristic adjustments as heuristic learning

The presented heuristic adjustments are simple examples of HL. In this case, the features vector of a state s consists only of a single number: $h(s)$. Each type of adjusted heuristic can be constructed by ML using a proper loss function. E.g., when we use MAE, the learned heuristic will be close to h^{avg} as h^{avg} has the lowest possible MAE on the set of training data, etc.

Beyond point estimates

Historically, researches have been developing heuristics that provide point estimates of lower bounds because they can be created easily using some form of relaxation ([75]). Search algorithms reflect this and vast majority of them uses heuristics as point estimators. This holds for A^* and it's variants, as well as for greedy heuristic search, beam stack search and many others.

There is however a much larger variety of information that the heuristic could provide.

1. standard lower bound: $\forall s : h(s) = b$, such that $b \leq h^*(s)$
2. upper bound: $\forall s : h(s) = t$, such that $h^*(s) \leq t$
3. interval: $\forall s : h(s) = [b, t]$, such that $b \leq h^*(s) \leq t$
4. confidence interval: e.g. for a 90%-confidence interval: $h(s) = [b, t]$, such that $\forall s : |\{s_i \in \Delta_s \mid b \leq h^*(s_i) \leq t\}| \geq 0.9 \cdot |\Delta_s|$
5. distribution characteristics: $h(s) = (\mu, \sigma^2)$, such that
   - $average[h^*(s_i) \mid s_i \in \Delta_s] \leq \mu$
   - $\mathrm{var}[h^*(s_i) \mid s_i \in \Delta_s] \leq \sigma^2$
6. density function estimate: $h(s) = q$, such that
   - $q : \mathbb{R} \mapsto [0, 1]$
   - $\forall x \in \mathbb{R} : \frac{|\{s_i \in \Delta_s | h^*(s_i) < x\}|}{|\Delta_s|} \leq q(x)$

It is difficult to design a standard heuristic (i.e. hand coded, not data-based) that would provide this kind of information. For example: providing an upper bound is equivalent to deciding whether the planning task is *solvable*. This alone is *PSPACE-hard* in the STRIPS formalism. (See [9], theorem 3.1.) Even if such heuristics existed, current search algorithm would not be able to utilize the additional information.

With the data-oriented approaches, however, it is possible to obtain this kind of more elaborate information, or at least its approximations. We believe that data-based approaches to heuristic design have a great potential and could lead to spawning a whole new generation of search algorithms that will be able to use the additional information and even provide some form of guarantees on run-time and solution quality.

In this thesis, we do not pursue the endeavor of creating new search algorithms. We use the HL approach to learn a heuristic in the standard format and we use a standard A^* algorithm to deploy it.

3. The Framework

In this chapter, we present our HL framework. We describe our approach to obtaining the training data, extracting features and selecting a loss function.

We work with the *Type II HL* (see section 1.3.1) and our approach is domain-independent in a sense that the domain from which the training problems come might be arbitrary. Once the model is trained on data from some specific domain, it will only be applicable to problems from that domain but the process can be repeated on any domain of interest. Our approach can therefore be described as an automatic creation of domain-specific heuristic for any domain.

Our approach is able to generalize over both *fluent* and *rigid* predicates. With respect to action-costs, our framework is applicable to *DDCosts* domains but not to *IDCosts* ones. See 1.3.2 for details. The technique is not applicable to *Type III HL*, i.e., it can't generalize across domains.

We are given problems $P_1, P_2, \dots P_k$ from the same domain $\mathcal{D}$. In the first phase, we train an ML model based on training data generated from given problems. We call this phase the *domain-analysis phase*. After the model is trained, new, previously unseen problem P from the same domain $\mathcal{D}$ is given and solved by A^* using the trained model as heuristic. We refer to this phase as the *deployment phase*. We focus here on the domain-analysis phase as in the deployment phase we use standard A^* algorithm without any modifications.

3.1 Training Data

The training data are pairs $(f(s_i), h^*(s_i))$ where s_i are states, $f(s_i)$ denotes features of s_i and $h^*(s_i)$ is cost-to-go from s_i to the nearest goal state which plays the role of a *target* during the training.

In typical ML applications the training data are given as inputs. In the HL scenario, the situation is somewhat different as it is possible to label the data without the need of human assistance. The training data can be generated and labeled automatically given enough computation power. This is the preferred way of acquiring the training data in HL since large datasets of goal-distances of states are not readily available for most planning domains.

Generating the training data involves three tasks: sampling states s_i from $S^{P_1} \cup S^{P_2} \cup \dots \cup S^{P_k}$, computing features $f(s_i)$ and calculating goal-distances $h^*(s_i)$ for those states.

Schema of the analysis phase is depicted in figure 3.1

3.1.1 Sampling the state spaces

We generate the training data ourselves hence we have a direct control over its properties. From the ML perspective, it is important that training data come from the same probability distribution as the data encountered during the deployment. Evaluating the model on completely different type of inputs would

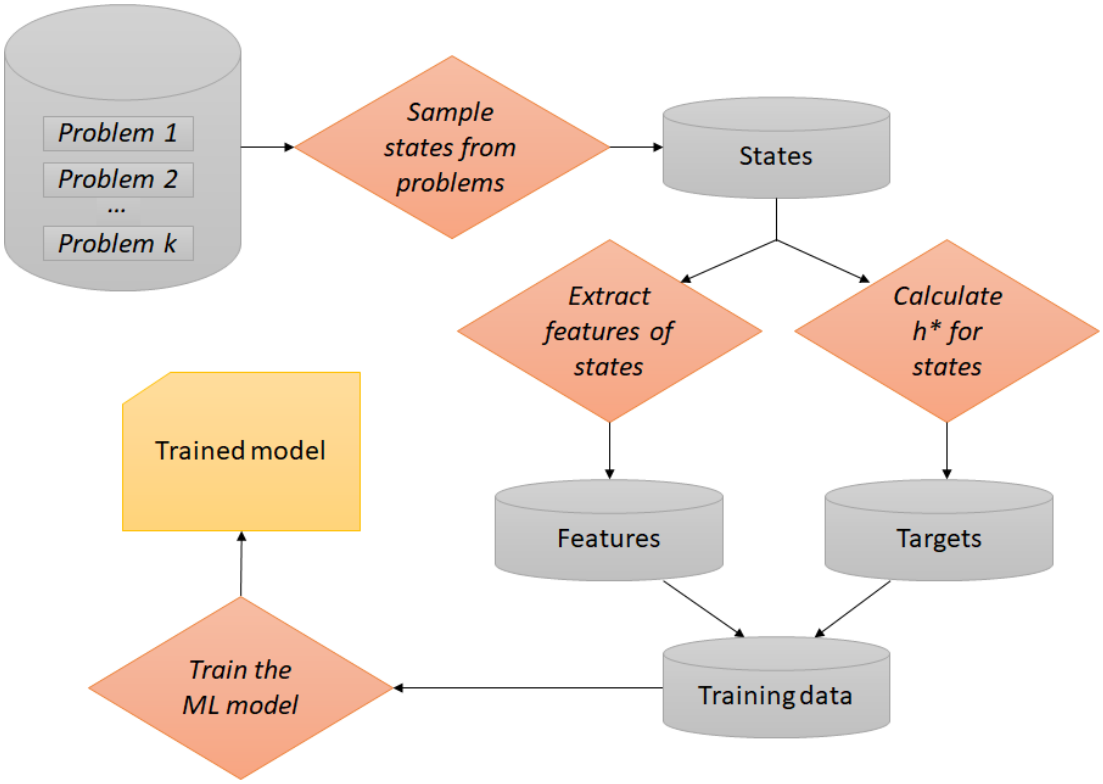

Figure 3.1: Schema of the domain-analysis phase.

lead to poor performance and unpredictable behaviour, especially when using a *high variance*[1] model such as deep neural network (as reported e.g. in [24]).

Computing $h^*(s_i)$ is very time-consuming hence many existing approaches only use states for which it is easy to compute $h^*(s_i)$, such as states close to goal or states that lay on optimal paths from $s_0^{P_j}$ to goal state. This however violates the distribution requirement.

We would like to train the model on the same type of states that it will encounter during the search phase. This would require to predict what kind of states will A* expand. Making such predictions is tricky as the set of expanded nodes depends on the heuristic used which depends on how the model is trained and that depends back on the choice of training data.

We adopt a technique by [1], which solves this issue by an iterative procedure that combines training and search steps.

The first set of training samples is empty. Then in each iteration the model is trained on current set of samples and a time-limited search is performed on all training problems using the trained model as a heuristic. States that the algorithm expanded are collected and used as training samples in the next iteration. The time limit is increased in each successive iteration, and the process continues until sufficient amount of samples is generated or all training problems can be solved within the time limit. In the first iteration where the set of training samples is empty, the model is not actually trained but a blind heuristic is used instead.

Pseudocode for this process is presented as Algorithm 1.

[1]See Bias–variance tradeoff. E.g. https://en.wikipedia.org/wiki/Bias-variance tradeoff

Algorithm 1: Domain analysis phase

Input: Set of planning problems $\{P_j\} \subset \mathcal{D}$ used for training
features extractor F
Output: Trained model M that realizes mapping $F[S^{\mathcal{D}}] \mapsto \mathbb{R}$

$L := 1$;
trainingStates $:= \emptyset$;
repeat
  compute $h^*(s_i)$ for each state $s_i \in$ *trainingStates*;
  assign features $f_i = f(s_i)$ to all states $s_i \in$ *trainingStates*;
  $M :=$ train neural net on data $\{(f_i, h^*(s_i))\}$;
  foreach *problem* $P \in \{P_j\}$ **do**
    run IDA* on P with time limit L minutes using M as heuristic;
    $T :=$ states of P that were expanded during the search;
    trainingStates $:=$ *trainingStates* $\cup\, T$;
  end
  $L := L + 1$;
until *termination criterion is met*;
return M;

3.1.2 Calculating goal-distances

Goal-distances of states can be calculated by a simple *BFS* or A^* with an admissible heuristic. This process can take minutes or even hours and since we work with millions of training states, this approach is not practical.

We decided to use ad-hoc solvers to calculate goal-distances for sampled states. This allows us to work with larger training problems in reasonable time. The ad-hoc solvers we've developed are briefly described in attachment A.2.

The usage of ad-hoc solvers is not necessary and they don't limit applicability of our method in any way. They can be omitted if the user has sufficient computational power to calculate goal-distances directly or if they have an alternative source of training data.

The issue of obtaining well-distributed training data in reasonable time is currently being extensively studied ([24, 67, 4, 52]). In this thesis, we don't investigate sampling techniques. We circumvent the issue by using the bootstrapping technique together with our ad-hoc solvers and we focus on different aspects of HL.

Imperfect goal-distances

Ad-hoc solvers that we use don't guarantee optimality of solutions they provide. Instead they find near-optimal solutions in reasonable time. We can however still use these imperfect values to train the network. Solvers may both over- and under-estimate the true distance-to-go. This can be seen as a random noise in the training data.

Such noise is often present in ML applications and the ML models are made to be able to handle it. Given enough data, the ML model will be able to distinguish signal from noise, i.e., it will be able to recognize and partially remove the noise in training data and provide even better estimates than the ad-hoc solvers used for

training. Imperfect goal-distances were used as labels in [67, 23] where authors report that the learned heuristic indeed outperforms its teacher.

3.2 Features Engineering

Features extractor assigns real-numbered vectors of fixed size to states. This is one of the most important components in any ML application[2].

Given a planning domain $\mathcal{D}$, the features extractor F realizes a mapping $F : S^{\mathcal{D}} \mapsto \mathbb{R}^{\lambda(\mathcal{D})}$, i.e., assigns a real valued vector to any state of any problem from the domain of interest where $\lambda(\mathcal{D})$ is the size of feature vectors for domain $\mathcal{D}$.

3.2.1 Required properties

Requirements on the features extractor are closely related to the generalization capabilities that we expect from the model. We work with the *Type II HL* and we use one-to-one model (see section 1.2.4) hence length of the feature vector needs to be *fixed* and independent of the specific problem instance.

Features should be *informative* in a sense that states with different goal-distances should have different features, and *comparable* among problems from the whole domain so that the knowledge is transferable to previously unseen problems. We would also like features to be *invariant under objects renaming*. I.e. if we changed names of constants in the PDDL representation of the state, the new state should have the same features as the original.

Properties of P that affect $h^*(s^P)$ have to be accessible to the model. Namely the set of available actions and the goal condition. The model is trained on problems from a single domain and for such problems the set of actions is always the same so it is not necessary to encode it into features of states. Goal conditions, however, must be encoded so that the learned knowledge is transferable to problems with different goal states.

3.2.2 Simple features

We will review here two popular features extraction techniques used for HL.

SAS^+ representation as features

In the SAS^+ formalism, states are represented as vectors of values of state variables, i.e., vectors of integers. It is possible to use this representation as the feature vector.

This features extractor is very informative as it assigns unique vector to each state. I.e. it will never happen that two states have the same features but different targets which typically happens with standard features extractors. Also, features can be computed in zero time as we already have states represented as "feature vectors" during the planning process.

Size of the features vector in this case depends on the number of objects in the planning problem hence it can't generalize over problems of different size. It

[2]https://towardsdatascience.com/machine-learning-isnt-models-it-s-features-be87b386db39

cannot even generalize over problems of the same size that have different goal conditions since the goal conditions are not represented in the vector. For these reasons, the SAS^+-features vector is only suitable for the *Type I HL*.

Heuristics as features

Given a set of simple heuristics $B = \{h_1, h_2, \ldots, h_q\}$. We can calculate features as $F^B(s) = (h_1(s), h_2(s), \ldots, h_q(s))$. HL papers often use a set of *pattern database heuristics* (PDBs). See [75].

This features extractor guarantees that vectors have fixed size and it is relatively well informed. Heuristics take into account goal conditions hence the features are comparable among problems with different goals, at least to some degree.

Heuristics can be evaluated on problems of any size so the features extractor can in principle be applied to problems of different size and its computation is quite fast as long as the heuristics are simple enough. Due to these properties, it is currently one of the most popular features extractors for HL.

There are however two issues with this approach.

PDBs are not transferable between problems. PDB is based on a pattern i.e. a set of objects from the planning problem. When a set of objects is selected and a *pattern* is created, there is no guarantee that new problems will contain objects with the same names, and even if they do, the role of those objects might be different. I.e. it is not invariant under objects renaming.

Also, goal conditions of new problems might differ from those on which the pattern has been created. Even if we re-computed the PDB with the same pattern on the new problem, those two values would not be comparable.

Another issue rises from the fact that accuracy of a heuristic often deteriorates as the problem size increases. See the following example.

Example 3.1

> There are two problems P_1, P_2 with initial states $s_0^{P_1}, s_0^{P_2}$ respectively. The first problem is small, it has $h^*(s_0^{P_1}) = 20$ while the second is larger, we have $h^*(s_0^{P_2}) = 100$. We have an admissible heuristic h. Let's find two states $s_1^{P_1}$ and $s_2^{P_2}$ such that $h(s_1^{P_1}) = h(s_2^{P_2}) = 15$.
>
> Typically, the *relative* accuracy of a heuristic much higher on small problems (see [75] section 3.5.2), i.e. we have $h^*(s_1^{P_1}) = 20$ while $h^*(s_2^{P_2}) = 40$. When used in HL, these two states would have the same features: $\langle 15 \rangle$ but very different targets: 20 vs. 40. This illustrates that the features are not comparable among problems of very different size.

Heuristic values can be used as features in the *Type I HL* where none of these issues occur. The approach is also useful for learning and efficient combination of several heuristics.

We believe that *Type II HL* system requires a much more flexible features extractor as features of states of different problems must be comparable. We have developed a new way of extracting features that is designed specifically for planning applications. We will describe it in chapter 4.

3.3 Error Function

Error function, or *loss function* is the criterion that is minimized during the training. In the regression setting, MSE is the most popular loss function but other loss functions also exist. MSE for 1D output is defined as

$$MSE = \frac{\sum(y_i - \hat{y}_i)^2}{n}$$

where y_i is the expected output for the i-th sample, $\hat{y}_i$ is the actual output of the model on the i-th ($y_i, \hat{y}_i \in \mathbb{R}$) sample and n in the number of samples. The sum goes over all samples.

In the heuristic learning scenario, the expected output is goal-distance of the state, i.e. $y_i = h^*(s_i)$ and the trained model is used as a heuristic function, so we can denote $\hat{y}_i = h_L(s_i)$, where h_L is the resulting learned heuristic. By using this notation it is clear that MSE criterion minimizes average of $(h^*(s_i) - h_L(s_i))^2$ which is average of $R_{h_L}(s_i)^2$ - squared residues. (See definition 17.)

3.3.1 Choice of the Loss Function

Choice of the loss function has a significant impact on properties of the learned heuristic as explained in detail in section 2.3.2. For example, we could change the formula for MSE like this:

$$MSE^{(w)} = \frac{1}{n}\sum_{i=1}^{n} err^{(w)}(y_i, \hat{y}_i)$$

where

$$err^{(w)}(y_i, \hat{y}_i) = \begin{cases} w \cdot (\hat{y}_i - y_i)^2 & if\ \hat{y}_i > y_i \\ (\hat{y}_i - y_i)^2 & otherwise \end{cases}$$

for some $w > 1$.

This would penalize overestimation more harshly than underestimation and hence should make the resulting heuristic closer to being "admissible". This attempt has been made in [93].

The loss function affects properties of the resulting heuristic. It is however not clear what the *desired* properties are, as discussed in section 2.3.1. User preferences must be taken into account when setting properties of the learned heuristic with respect to the search speed vs. solution quality.

We experimented with two loss functions: the standard *MSE* and an adjusted version that we call *LogMSE* defined as follows.

$$LogMSE = \frac{\sum\left[\log(y_i + 1) - \log(\hat{y}_i + 1)\right]^2}{n}$$

for $y_i, \hat{y}_i \geq 0$.

Comparison of heuristics learned by *MSE* and by *LogMSE* can be found in section 5.4.5. Advantages of *LogMSE* in the HL scenario are discussed in detail in section 5.5.

4. Graph-based Features

This chapter presents our novel way of extracting features of states of planning problems. We use a direct encoding of the state to an integer vector without relying on existing hand-coded heuristics.

Our method is based on transforming the state to a graph which we call *Object graph* and then counting the number of occurrences of specific subgraphs in the object graph and using those counts as features. We first define the object graph.

4.1 Object Graph

We work with the PDDL representation of the problem, so we are given a set of constants C, a set of predicate symbols Ψ, a set Q of all *instantiated predicates* and a goal condition $G \subset Q$. We don't support negative goal conditions as they can be compiled-away so the goal condition is just a set of predicates that need to hold.

Let's first define the notion of *vertex labeled graph*.

Definition 20 (Vertex labeled graph)**.** *A* ***vertex labeled graph*** *is a triple* $(V, E, \mathcal{L})$ *where* (V, E) *is a graph and* $\mathcal{L} : V \mapsto \mathbb{N}^0$ *is a labeling function.*

Definition 21 (Object graph)**.** *An* ***object graph*** *for a state s of a planning problem P (denoted $G(s^P)$) is a vertex labeled graph $G(s^P) = (V, E, \mathcal{L})$ defined as follows. The set of vertices V consists of four disjoint sets:*

1. *there is a vertex v_c for every constant $c \in C^P$*
2. *there is a vertex v_ψ for every predicate symbol $\psi \in \Psi^P$*
3. *there is a vertex v_q for every instantiated predicate $q \in Q^P$ that is true in s*
4. *there is a vertex v_q^g for every goal predicate $q \in G^P$*

The set E contains an edge $e_{q\psi}$ from v_q or v_q^g to v_ψ if instantiated predicate q uses the predicate symbol ψ, and an edge e_{cq} from v_c to v_q or v_q^g if instantiated predicate q contains constant c.

Every vertex of the graph is labeled by an integer using a labelling function $\mathcal{L} : V \mapsto \mathbb{N}^0$ that looks as follows.

- *every constant-vertex v_c is labeled* 0
- *every predicate-vertex v_q is labeled* 1
- *every goal-predicate-vertex v_q^g is labeled* 2
- *every predicate-symbol-vertex v_ψ is assigned a unique label from $[3, 4, \ldots, |\Psi| + 2]$ where $|\Psi|$ is the number of predicate symbols.*

If types are present, they are treated as unary predicate symbols. E.g. if a constant *plane1* has type *Plane*, we add a predicate symbol *Plane* to Ψ and we set the instantiated predicate $Plane(plane1)$ to be true in all states.

Example 4.1

Let's take a look at the initial state of the problem *pfile1* of the *zenotravel* domain. (See A.1.1 for details about the domain and URLs to the PDDL representation.) *Zenotravel* domain uses the following predicate symbols:

```
at, in, fuel-level, next
aircraft, person, city, flevel
```

Predicate symbols on the first line are binary, the others are unary (types). Problem *pfile1* contains the following constants:

```
plane1,
person1, person2,
city0, city1, city2,
fl0, fl1, fl2, fl3, fl4, fl5, fl6
```

The initial state is defined by these fluent predicates:

```
at(plane1, city0)
fuel-level(plane1, fl1)
at(person1, city0)
at(person2, city2)
```

and the following rigid predicates:

```
aircraft(plane1),
person(person1), person(person2),
city(city0), city(city1), city(city2),
flevel(fl0), flevel(fl1), flevel(fl2), flevel(fl3),
    flevel(fl4), flevel(fl5), flevel(fl6),
next(fl0, fl1), next(fl1, fl2), next(fl2, fl3),
    next(fl3, fl4), next(fl4, fl5), next(fl5, fl6)
```

Goal condition is as follows:

```
at(plane1, city1)
at(person1, city0)
at(person2, city2)
```

In figure 4.1 there is the object graph for this state. Colors represent labels. Constants are colored red (label 0), initial predicates are green (label 1) and goal predicates yellow (label 2). There are 8 predicate symbols that are represented by 8 nodes with mutually different colors. These have labels 3 - 10. Unary predicate symbols are drawn as circles, binary ones as diamonds.

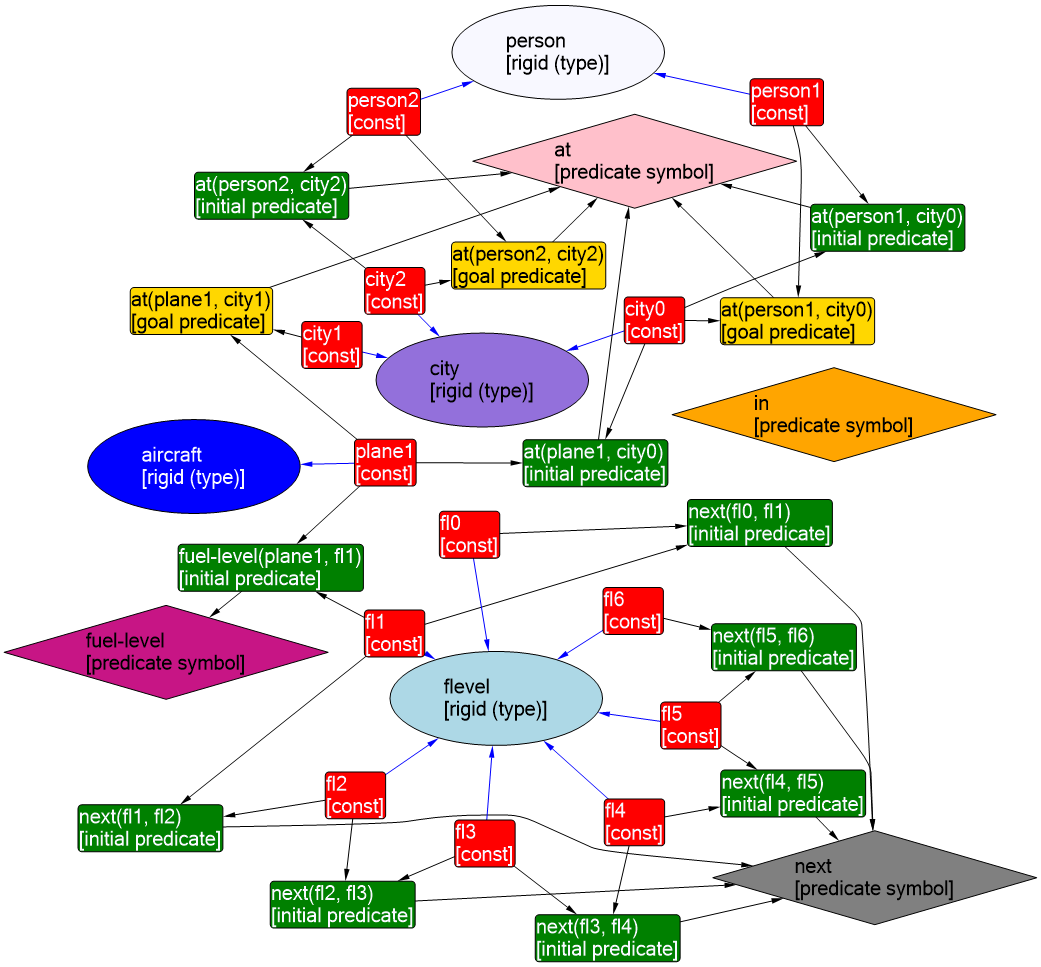

Figure 4.1: Example of an object graph for the initial state of problem *pfile1* of the *zenotravel* domain.

4.1.1 Properties of the graph

Size of the graph $G(s^P)$ is linear in the size of s and in the number of goal conditions. The number of vertices is $|C|+|G|+|s|+|\Psi|$. Every predicate (initial or goal) is connected to the corresponding predicate symbol and a constants where a is the arity of the predicate symbol. Hence, number of edges is at most $(a_m + 1) \cdot (|s| + |G|)$ where a_m is the maximum arity of some predicate symbol from Ψ, $|s|$ is the number of predicates that are positive the s and $|G|$ is the number of goal predicates.

The graph need not be connected as seen in the example, but it often is. the number of constants, initial and goal predicates is problem-specific. Number of predicate symbols, on the other hand, is domain-specific and will be the same for all problems within the domain.

The object graph is an equivalent representation of the state and goal condition. The original PDDL representation of the state and of the goal can be reconstructed back from the graph.

4.2 Extracting Features

We use the object graph to extract features by counting the number of occurrences of specific subgraphs while taking into account labels of nodes. Let's first define the necessary notions.

Definition 22 (Isomorphism of labeled graphs)**.** *Let $G_1 = (V_1, E_1, \mathcal{L}_1)$ and $G_2 = (V_2, E_2, \mathcal{L}_2)$ be two vertex labeled graphs. We say that G_1 and G_2 are* ***isomorphic*** *if $|V_1| = |V_2|$ and there exists a bijection $f : V_1 \mapsto V_2$ such that*

1. $\forall v_1, v_2 \in V_1 : (v_1, v_2) \in E_1 \Leftrightarrow (f(v_1), f(v_2)) \in E_2$
2. $\forall v \in V_1 : \mathcal{L}_1(v) = \mathcal{L}_2(f(v))$

Let B^k_α be a set of all connected non-isomorphic vertex-labeled graphs containing at most α vertices where the labels are from $\{0, 1, 2 \ldots k-1\}$.

Example 4.2

Figure 4.2 shows an example of graphs B^2_2 - colors represent labels.

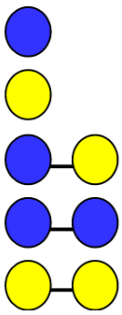

Figure 4.2: The set of graphs B^2_2.

Figure 4.3 shows the set B^2_3.

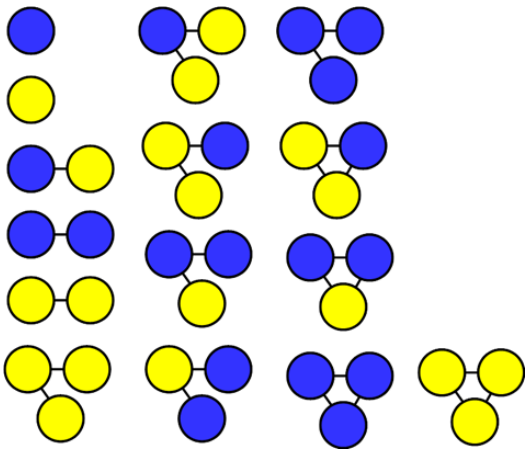

Figure 4.3: The set of graphs B^2_3.

Definition 23 (Occurrence of a graph in another graph)**.** ***Occurrence*** *of a graph G_1 in graph G_2 is a set of vertices T of G_2 such that induced subgraph of G_2 on T is isomorphic to G_1.*

Definition 24 (Object graph features)**.** *Given a state s and $\alpha, k \in \mathbb{N}$, the* ***object graph feature vector*** *of s (denoted $F_\alpha(s)$) is an integer vector of size $|B_\alpha^k|$ whose i-th component is the number of occurrences of the i-th graph from B_α^k in $G(s)$.*

Example 4.3

Consider the graph G in figure 4.4 with two different labels represented by colors. (I.e. the parameter k equals two.) We use $\alpha = 2$, i.e. we count occurrences of connected subgraphs of size up to 2. The set B_2^2 is shown in figure 4.2.

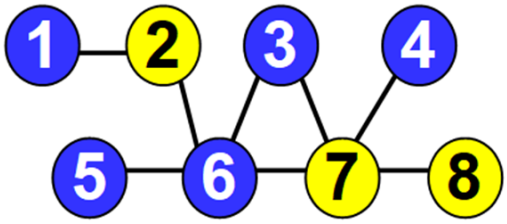

Figure 4.4: A simple graph used to demonstrate computation of features.

We order elements of B_2^2 to a sequence and fix their order. The order is arbitrary but must be the same for all states that we assign features to. Then we count the number of occurrences of each element of B_2^2. The resulting vector is $F_2(G) = (5, 3, 5, 2, 1)$ as shown in figure 4.5.

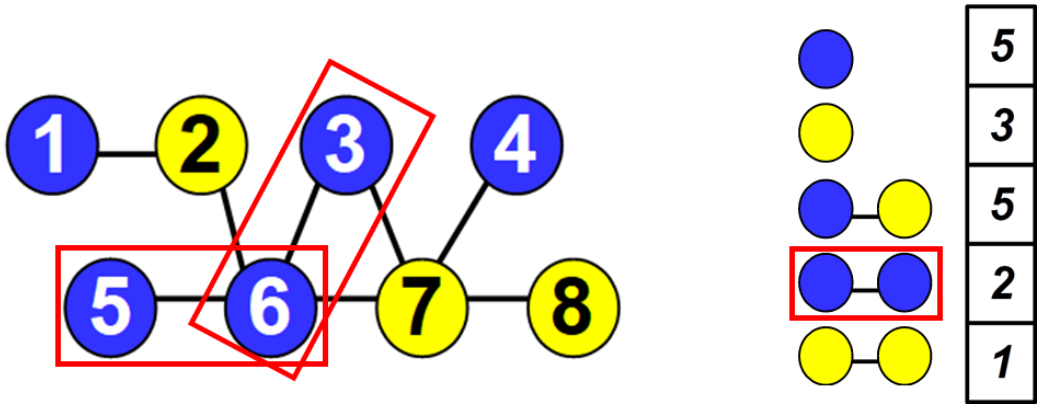

Figure 4.5: Example of extraction of features from a vertex labeled graph. Left-hand side: example graph, right-hand side: set of B_2^2 together with the resulting features vector. The two occurrences of one of the graphs are marked.

4.3 Properties of $F_\alpha(s)$

The parameter α of $F_\alpha(s)$ determines the maximum size of subgraphs that are considered and it can be adjusted by the user. With low α, the vector will be short and will contain less information about the state but its computation will be faster, and vice versa.

The parameter k in the definition of Object graph features (definition 24), on the other hand, cannot be adjusted by the user and is fully determined by the number of labels used in the object graph of s, i.e. by the domain. We've intentionally defined the labeling function in such a way that k is *domain-specific* but not *problem-specific*. As stated in definition 21, the number of labels equals to the number of predicate symbols + 3. This ensures that size of the features vector is constant across the whole domain and hence our features extraction is usable for the *Type II HL*. See section 1.3.1 for details.

4.3.1 Extracted knowledge

The number of occurrences of small graphs with size 1 and 2 captures knowledge about size of the problem: the number of objects for each type, the number of positive predicates and size of the goal condition. This information has been used before in hand-coded HL features extractors ([34]) and is known to be beneficial.

If the domain only contains unary and binary predicates, graphs with size 5 can capture knowledge about the number of not accomplished goals. The graph looks as depicted in figure 4.6.

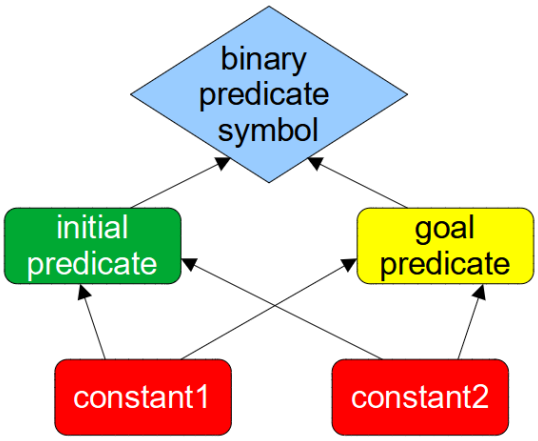

Figure 4.6: A graph of size 5 which indicates that a binary predicate is already accomplished.

Each occurrence of the graph represents a goal predicate that is already accomplished in the current state. The total number of goal predicates is captured by the number of occurrences of another graph so the model can calculate the value of $h_{GoalCount}$, for example. (See section 1.1.5 for description of $h_{GoalCount}$).

In *zenotravel* domain, this knowledge can be captured even by graphs of size 4 as the predicate symbol-vertex is not relevant. This tells the model how many passengers and planes are not yet at their destinations. A larger graph can for example represent a pair of passengers that can share the ride, i.e. having the same current location and the same destination, etc.

In *blocks* domain (see A.1.2), graphs of size 4 can tell the number of blocks that are not yet correctly placed. Occurrence of a certain graph of size 5 can for example indicate the fact, that there are 2 blocks A on top of B and B is not correctly placed so both of them will have to be moved, etc.

In general, the knowledge that the features vector captures is domain-specific. The larger subgraphs we count the more complex knowledge the vector can capture.

4.3.2 Length of the vector

Length of the features vector is limited by the size of B_α^k which grows exponentially in both α and k. See [3] for an asymptotic formula for the number of connected graphs.

Length of the vector is at most $\sum_{i=1}^{\alpha} 2^{\frac{i(i-1)}{2}} C'_i(k)$, where $2^{\frac{i(i-1)}{2}}$ is the number of graphs on i vertices and $C'_i(k) = \frac{(k+i-1)!}{i!(k-1)!}$ is the number of combinations with repetition of size i from k elements, i.e. the number of ways how labels can be assigned to vertices.

In practice, $|F_\alpha(\cdot)|$ is much lower since we only include graphs from B_α^k that **occurred at least once** in the training data. Vast majority of graphs do not occur due to the way how the object graph is defined. For example, every predicate symbol ψ_i has its own label l_i and every object graph contains exactly one vertex with such label. Graphs that contain more than one vertex labeled l_i can never occur. Also, graphs that contain edge between vertices with the same label cannot occur, etc.

The set of graphs whose occurrences we actually count we call the **Features graphs**. The set of features graphs is a subset of B_α^k. In the rest of this chapter, we use B_α^k also when referring to the set of Features graphs as there should be no confusion.

Actual lengths of vectors for various values of α are shown in Table 5.1 in the *Experiments* section.

4.3.3 Expressive power

We can measure expressive power of features extractor by calculating the relative amount of states that are assigned the same features event though they have different targets. We say that the features extractor has *high expressive power* if amount of such states is low. E.g. an extractor that assigns the same features to all states has the lowest possible expressive power while an extractor which assigns a unique vector to every state has the highest possible expressive power.

ML models produce their predictions solely based on features of states hence they require that states with different cost-to-go are distinguished by having different features. This is important especially among states whose difference in cost-to-go is high. On the other hand, if the expressive power of the features extractor is too high, it harms generalization capabilities of the model.

Our graph-based features extractor guarantees that features will be unique as long as the size of subgraphs that we count is greater or equal to size of the object graph, i.e. when $\alpha \geq n$ where n is size of the object graph. This is more of a theoretical property as in practice α is quite low (e.g. $2 \sim 6$) while size of object graphs as much larger ($10 \sim 1000$) and potentially unlimited.

It is not known whether or not $F_\alpha(G)$ uniquely determines every graph on n vertices for some $\alpha < n$, not even for $\alpha = n - 1$. It is an open problem in graph

theory known as the *Reconstruction conjecture*[1].

In our application, α is always fixed before the training and we require that the trained model is applicable to problems of any size. Hence we cannot guarantee that $\alpha \geq n$ as the size of object graphs could be arbitrarily large. For this reason features are not unique for all states.

The expressive power depends on α: the higher α the longer and more expressive the vectors are. It is difficult to provide any theoretical guarantees as the expressive power is domain dependent. We've tried to measure it experimentally on the set of training data we've collected. Results can be found in section 5.3.

Longer vectors always contain strictly more information than the shorter ones. For any α and any G, the set of graphs whose occurrence F_α counts is a subset of the graphs that $F_{\alpha+1}$ counts hence the features vector $F_\alpha(G)$ is contained in the vector $F_{\alpha+1}(G)$.

4.4 Computing Features from Scratch

Given an object graph $G(s)$, the features vector $F_\alpha(s)$ can be computed by a recursive procedure that iterates through all connected subgraphs with size up to α in the graph and its time complexity is proportional to the number of such subgraphs. It is difficult to estimate the number in general as it strongly depends on the structure of the graph. E.g. a cycle with n vertices contains just n connected induced subgraphs of size $\alpha < n$, while a clique on n vertices contains $\binom{n}{\alpha}$ such subgraphs. The number depends on the edge-connectivity of the graph, i.e. on how many paths are there between pairs of vertices, as well as on degrees of vertices.

Experimental results show that the time required to compute features grows exponentially in α as well as in the size of object graph. The size of object graph is proportional to the number of objects in the planning problem, hence the method is very slow on larger problems.

4.5 Computing Features Incrementally

When we take into account the way how A^* algorithm operates, it is possible to calculate the features vector incrementally. A^* expands nodes in a forward manner. Roughly speaking, the algorithm operates in the following way:

1. select state s from the open list
2. enumerate all its successors s_i
3. for each successor compute its heuristic estimate and insert it to the open list

After computing $F_\alpha(s)$, we can store this information and utilize it later when calculating features of its successors $s_i = \gamma(s, a)$. States s and s_i differ only locally as each action a changes just a small number of predicates. Knowing this, we can only modify counts of subgraphs that were affected by applying the action.

[1] https://en.wikipedia.org/wiki/Reconstruction_conjecture

Unfortunately, $F_\alpha(s)$ and a alone are not sufficient to determine features of the successor. For any fixed α there exist states $s_1 \neq s_2$ and action a_1 applicable to both states such that $F_\alpha(s_1) = F_\alpha(s_2)$ but $F_\alpha(\gamma(s_1, a_1)) \neq F_\alpha(\gamma(s_2, a_1))$. A more sophisticated approach is needed.

4.5.1 Extended object graph

We have come up with a way to generate features incrementally using so called *Extended object graph*. Lets first define the required notions.

Definition 25 (Contribution of a vertex)**.** *Let $G(s)$ be an object graph and v its vertex.* ***Contribution*** *of v to $F_\alpha(s)$ (denoted by $C(v)$) is a set of all occurrences of graphs from B_α in $G(s)$ which intersect with v.*

Occurrence is a set of vertices of $G(s)$ as stated in Definition 23.

Definition 26 (Extended object graph)**.** ***Extended object graph*** *(EOG) is a structure associated with a state s that contains:*

- *object graph $G(s)$*
- *vector $F_\alpha(s)$*
- *contribution $C(v)$ for every vertex $v \in G(s)$*

Applying action to a state s can be viewed as performing some local changes in $G(s)$. These changes can be decomposed into a sequence of several atomic operations of 3 types: *AddVertex*, *RemoveVertex* and *AddEdge*.

Example 4.4

> In Zenotravel, there is an action $a = board(person1, city1, plane1)$ that removes predicate $at(person1, city1)$ adds predicate $in(person1, plane1)$. Given $G(s)$, we can construct $G(\gamma(a, s))$ by first removing vertex that represents predicate $at(person1, city1)$, then adding vertex for predicate $in(person1, plane1)$ and then successively adding edges between the new vertex and vertices representing in, $person1$ and $plane1$.

We will now show how each of the three atomic operations can be performed incrementally over the extended object graph. Pseudo-code for each procedure is presented as Algorithm 2, 3 and 4 respectively.

We store the EOG in a carefully designed data structure (DS) that helps to speed-up the three operations. Our DS contains the following:

- representation of the original object graph
- every vertex of the graph holds reference to all occurrences that intersects with it
- every $r \in C(v)$ holds reference to the graph from B^k_α whose occurrence r indicates.

Algorithm 2: Remove vertex

Input: Extended object graph W, its vertex v
Output: Extended object graph W with vertex v removed

```
foreach occurrence r ∈ C(v) do
    foreach v_i ∈ r do
        remove r from C(v_i);
    end
    let j be index of graph in B_α^k whose occurrence r holds;
    decrease value of F_α(s)[j] by 1;
end
remove vertex v from G(s) together with all incident edges;
```

Algorithm 3: Add vertex

Input: Extended object graph W, new vertex v
Output: Extended object graph W with vertex v added

```
add vertex v to G(s);
let i be index of graph in B_α^k that consists of a single vertex with label
  same as v;
increase value of F_α(s)[i] by 1;
set C(v) = {{v}};
```

- every graph $b_i \in B_\alpha^k$ holds reference to all its occurrences in $G(s)$

There is also additional information stored in the *Features extractor* itself:

- for every graph $b_i \in B_\alpha^k$ and for every pair of vertices $v_1, v_2 \in b_i$ that are not connected by an edge, the result of adding an edge between v_1, v_2 is precomputed and index of the resulting graph in B_α^k is stored
- the same for connecting two graphs from B_α^k with an edge

See the following example.

Example 4.5

Let's use the graph in figure 4.7 to demonstrate. Numbers identify vertices, colors represent labels. The example should resemble an actual object graph: it is connected, there is relatively small number of labels and there are no edges between vertices with the same label. Colors red, gold and green resemble predicate-symbol labels (see definition 21), and there is exactly one vertex with such label in every Object graph. Blue color resembles predicate label.

Algorithm 4: Add edge

Input: Extended object graph W, two its vertices v_1, v_2
Output: Extended object graph W with edge (v_1, v_2) added

add edge (v_1, v_2) to $G(s)$;
foreach *occurrence* $r \in C(v_1) \cap C(v_2)$ **do**
    replace r in both $C(v_1)$ and $C(v_2)$ by occurrence of a graph with the same vertex set and edge (v_1, v_2) added;
    modify the vector F_α accordingly;
end
foreach *occurrence* $r_1 \in C(v_1) \setminus C(v_2)$ **do**
    foreach *occurrence* $r_2 \in C(v_2) \setminus C(v_1)$ **do**
        if $|r_1 \cup r_2| \leq \alpha$ **then**
            g = induced subgraph of $G(s)$ on vertices $r_1 \cup r_2$;
            add occurrence of g to both $C(v_1)$ and $C(v_2)$;
            let i be index of graph g in B^k_α;
            increase value of $F_\alpha(s)[i]$ by 1;
        end
    end
end

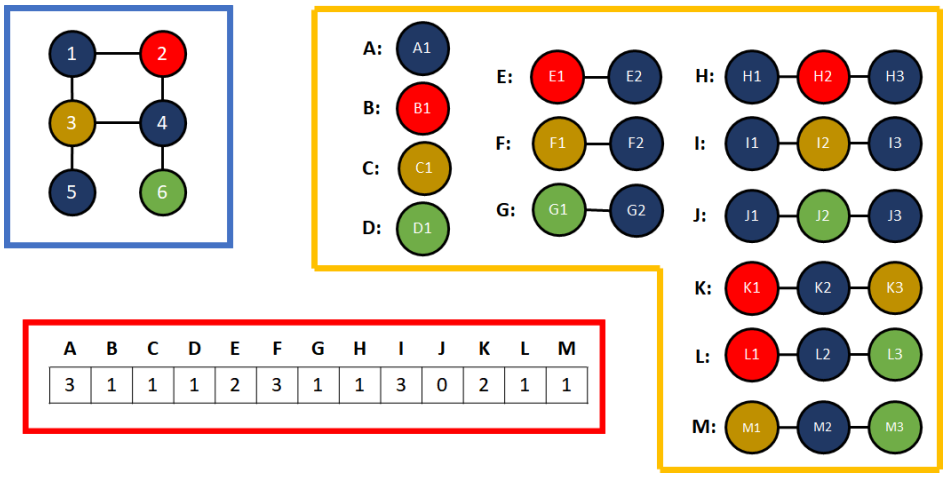

Figure 4.7: Example of a vertex labeled graph (top left), a set of 13 features graphs whose occurrences we count (on the right) and the corresponding features vector (bottom left).

The number of labels $k = 4$, and we count subgraphs of size $\alpha \leq 3$. The set of features graphs is depicted in figure 4.7. Only graphs that can actually occur are tracked. E.g. graphs with an edge between vertices with the same label or containing more than one vertex with red, green or gold label are not present.

Sum of the values in the features vector is 20 since there are 20 occurrences of the subgraphs we track. We denote them $r_1, r_2, \ldots, r_{20}$. Each occurrence is s set of vertices together with the corresponding isomorphism. Here is the list.

Occurrences of single-vertex graphs $A \sim D$:

- $r_1 = (A, \{A1 \mapsto 1\})$

- $r_2 = (A, \{A1 \mapsto 4\})$
- $r_3 = (A, \{A1 \mapsto 5\})$
- $r_4 = (B, \{B1 \mapsto 2\})$
- $r_5 = (C, \{C1 \mapsto 3\})$
- $r_6 = (D, \{D1 \mapsto 6\})$

Occurrences of two-vertex graphs: $E \sim G$:

- $r_7 = (E, \{E1 \mapsto 2, E2 \mapsto 1\})$
- $r_8 = (E, \{E1 \mapsto 2, E2 \mapsto 4\})$
- $r_9 = (F, \{F1 \mapsto 3, F2 \mapsto 1\})$
- $r_{10} = (F, \{F1 \mapsto 3, F2 \mapsto 4\})$
- $r_{11} = (F, \{F1 \mapsto 3, F2 \mapsto 5\})$
- $r_{12} = (G, \{G1 \mapsto 6, G2 \mapsto 4\})$

Occurrences of three-vertex graphs: $H \sim M$:

- $r_{13} = (H, \{H1 \mapsto 1, H2 \mapsto 2, H3 \mapsto 4\})$
- $r_{14} = (I, \{I1 \mapsto 1, I2 \mapsto 3, I3 \mapsto 4\})$
- $r_{15} = (I, \{I1 \mapsto 1, I2 \mapsto 3, I3 \mapsto 5\})$
- $r_{16} = (I, \{I1 \mapsto 4, I2 \mapsto 3, I3 \mapsto 5\})$
- $r_{17} = (K, \{K1 \mapsto 2, K2 \mapsto 1, K3 \mapsto 3\})$
- $r_{18} = (K, \{K1 \mapsto 2, K2 \mapsto 4, K3 \mapsto 3\})$
- $r_{19} = (L, \{L1 \mapsto 2, L2 \mapsto 4, L3 \mapsto 6\})$
- $r_{20} = (M, \{M1 \mapsto 3, M2 \mapsto 4, M3 \mapsto 6\})$

Note that the isomorphism need not be unique. E.g., in the occurrence r_{13}, we could have $r_{13} = (H, \{H1 \mapsto 4, H2 \mapsto 2, H3 \mapsto 1\})$ instead. This does not affect the results in any way.

Contributions of individual vertices look as follows:

- Vertex: 1, $C(1) = \{r_1, r_7, r_9, r_{13}, r_{14}, r_{15}, r_{17}\}$
- Vertex: 2, $C(2) = \{r_4, r_7, r_8, r_{13}, r_{17}, r_{18}, r_{19}\}$
- Vertex: 3, $C(3) = \{r_5, r_9, r_{10}, r_{11}, r_{14}, r_{15}, r_{16}, r_{17}, r_{18}, r_{20}\}$
- Vertex: 4, $C(4) = \{r_2, r_8, r_{10}, r_{12}, r_{13}, r_{14}, r_{16}, r_{18}, r_{19}, r_{20}\}$
- Vertex: 5, $C(5) = \{r_3, r_{11}, r_{15}, r_{16}\}$
- Vertex: 6, $C(6) = \{r_6, r_{12}, r_{19}, r_{20}\}$

The above information is stored in the EOG.

The following mapping is stored in the *Features extractor* and is shared by all EOGs. It holds precomputed results of adding edges between vertices of features graphs.

E.g., if there is an occurrence of the graph A on vertex 1 and an occurrence of the graph B on vertex 2 then adding an edge $(1, 2)$ will create a new occurrence of the graph denoted E on vertices $\{1, 2\}$. The isomorphism in this case is $\{E1 \rightarrow 1, E2 \rightarrow 2\}$.

The particular vertices $1, 2$ are not important as we can specify the isomorphism using just identifiers of vertices of graphs A and B. The generalized mapping is $\{A1 \leftrightarrow E2, B1 \leftrightarrow E1\}$. Vertices indicating where the edge is added can be expressed via graphs A and B as well. In this case, the new edge is $\{A1, B1\}$. This generalized information is shareable between all EOGs.

Following is the list of all *add-edge* operations that create new occurrence of some features graph.

- $\{A1, B1\} : E, \{A1 \leftrightarrow E2, B1 \leftrightarrow E1\}$
- $\{A1, C1\} : F, \{A1 \leftrightarrow F2, C1 \leftrightarrow F1\}$
- $\{A1, D1\} : G, \{A1 \leftrightarrow G2, D1 \leftrightarrow G1\}$
- $\{A1, E1\} : H, \{A1 \leftrightarrow H1, E1 \leftrightarrow H2, E2 \leftrightarrow H3\}$
- $\{A1, F1\} : I, \{A1 \leftrightarrow I1, F1 \leftrightarrow I2, F2 \leftrightarrow I3\}$
- $\{A1, G1\} : J, \{A1 \leftrightarrow J1, G1 \leftrightarrow J2, G2 \leftrightarrow J3\}$
- $\{B1, F2\} : K, \{B1 \leftrightarrow K1, F1 \leftrightarrow K3, F2 \leftrightarrow K2\}$
- $\{B1, G2\} : L, \{B1 \leftrightarrow L1, G1 \leftrightarrow L3, G2 \leftrightarrow L2\}$
- $\{C1, E2\} : K, \{C1 \leftrightarrow K3, E1 \leftrightarrow K1, E2 \leftrightarrow K2\}$
- $\{C1, G2\} : M, \{C1 \leftrightarrow M1, G1 \leftrightarrow M3, G2 \leftrightarrow M2\}$
- $\{D1, E2\} : K, \{D1 \leftrightarrow L3, E1 \leftrightarrow L1, E2 \leftrightarrow L2\}$
- $\{D1, F2\} : M, \{D1 \leftrightarrow M3, F1 \leftrightarrow M1, F2 \leftrightarrow M2\}$

We will new demonstrate the execution of *Remove vertex* and *Add edge* operations over the EOG.

Remove vertex 1

For each $r_i \in C(1)$, we remove r_i from all $C(v_i)$ sets that contain it. We also decrease the counter at the corresponding index in the features vector. The list of contributions now looks as follows:

- ~~Vertex: 1, $C(1) = \{r_1, r_7, r_9, r_{13}, r_{14}, r_{15}, r_{17}\}$~~
- Vertex: 2, $C(2) = \{r_4, $ ~~r_7~~$, r_8, $ ~~r_{13}~~$, r_{17}, r_{18}, r_{19}\}$
- Vertex: 3, $C(3) = \{r_5, $ ~~r_9~~$, r_{10}, r_{11}, $ ~~r_{14}~~$, $ ~~r_{15}~~$, r_{16}, $ ~~r_{17}~~$, r_{18}, r_{20}\}$
- Vertex: 4, $C(4) = \{r_2, r_8, r_{10}, r_{12}, $ ~~r_{13}~~$, $ ~~r_{14}~~$, r_{16}, r_{18}, r_{19}, r_{20}\}$

- Vertex: 5, $C(5) = \{r_3, r_{11}, \sout{r_{15}}, r_{16}\}$
- Vertex: 6, $C(6) = \{r_6, r_{12}, r_{19}, r_{20}\}$

Changes made to the features vector are depicted in figure 4.8.

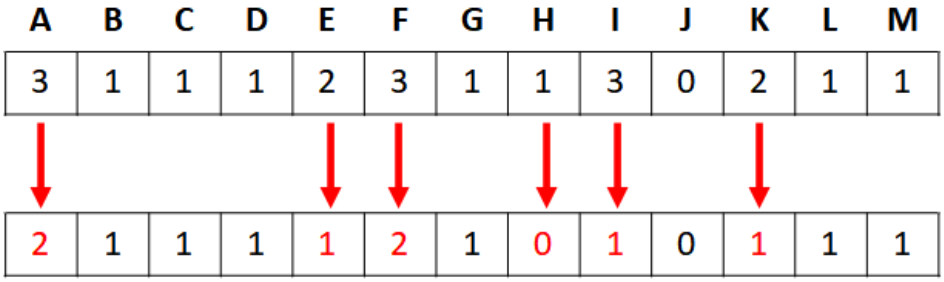

Figure 4.8: Changes to the features vector caused by removal of vertex 1.

Add edge $(5, 6)$

We add the edge to the original EOG depicted in figure 4.7.

First, we check occurrences that include both vertices 5 and 6. We would replace each such occurrence with an occurrence of a modified graph. No such occurrences are present in our example.

Now we check all pairs of occurrences (r_i, r_j) such that $r_i \in C(5)$, $r_j \in C(6)$ and the sum of sizes of $|r_i| + |r_j| \leq \alpha = 3$. In our example, we have 3 such pairs: (r_3, r_{12}), (r_{11}, r_6) and (r_3, r_6). For each pair, we identify which graph from the set of features graphs will newly occur after adding the edge. Here we use the precomputed mapping.

Processing pair (r_3, r_{12}): we are adding edge between vertices 5 and 6 that are mapped $A1 \mapsto 5$ and $G1 \mapsto 6$ in r_3, r_{12} respectively. We then access the precomputed mapping using index $\{A1, G1\}$, we get

$$(A1, G1) : J, \{A1 \leftrightarrow J1, G1 \leftrightarrow J2, G2 \leftrightarrow J3\}$$

This tells us that the new occurrence will look like this:
$r_{21} = (J, \{J1 \mapsto 5, J2 \mapsto 6, J3 \mapsto 4\})$. The concrete vertices are obtained by composing the precomputed mapping with mappings in r_3 and r_{12}.

After processing all three pairs, we have three new occurrences:

- $r_{21} = (J, \{J1 \mapsto 5, J2 \mapsto 6, J3 \mapsto 4\})$
- $r_{22} = (M, \{M1 \mapsto 3, M2 \mapsto 5, M3 \mapsto 6\})$
- $r_{23} = (G, \{G1 \mapsto 6, G2 \mapsto 5\})$

Contributions after adding the edge:

- Vertex: 1, $C(1) = \{r_1, r_7, r_9, r_{13}, r_{14}, r_{15}, r_{17}\}$
- Vertex: 2, $C(2) = \{r_4, r_7, r_8, r_{13}, r_{17}, r_{18}, r_{19}\}$
- Vertex: 3, $C(3) = \{r_5, r_9, r_{10}, r_{11}, r_{14}, r_{15}, r_{16}, r_{17}, r_{18}, r_{20}, r_{22}\}$
- Vertex: 4, $C(4) = \{r_2, r_8, r_{10}, r_{12}, r_{13}, r_{14}, r_{16}, r_{18}, r_{19}, r_{20}, r_{21}\}$

- Vertex: 5, $C(5) = \{r_3, r_{11}, r_{15}, r_{16}, r_{21}, r_{22}, r_{23}\}$
- Vertex: 6, $C(6) = \{r_6, r_{12}, r_{19}, r_{20}, r_{21}, r_{22}, r_{23}\}$

The features vector is updated as shown in figure 4.9.

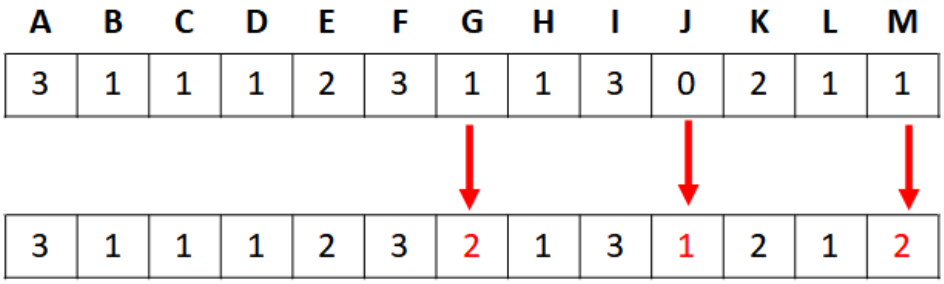

Figure 4.9: Changes to the features vector caused by adding edge $\{5, 6\}$.

Using this DS, we can perform operation *Add vertex* in $\mathcal{O}(1)$, *Remove vertex* in $\mathcal{O}(\alpha \cdot |C(v)|)$ and *Add edge* in $\mathcal{O}(\alpha \cdot |C(v_1)| \cdot |C(v_2)|)$.

EOG for the initial state is constructed from scratch and every other EOG is constructed from the EOG of its predecessor via applying a sequence of atomic graph operations that corresponds to the action used. The EOG is then stored together with each state when it is added to the open list.

To our best knowledge, this is the first features extraction technique developed for HL that allows incremental computation. With exception of SAS^+-based features, of course which don't require any computation.

5. Experiments

In this chapter, we present our main experimental results. We overview the data we've collected, compare performance of the learned heuristic with a baseline and analyze the results. We also present various statistics on results of the training, as well as performance of heuristic-adjustments.

We performed various statistical analyses and ML experiments using data generated from state-spaces of planning problems. Lets first take a look at the data we've collected.

5.1 Data

Our main goal is to perform the heuristic learning as described in chapter 3 so the data were collected to serve this purpose. We used the technique described in section 3.1.1 to obtain a set of states $\{s_j^{P_i}\}$. We then used our ad-hoc solvers to calculate estimates of $h^*(s_j^{P_i})$ for these states.

We perform the experiments on two domains: Zenotravel and Blocks. See attachments A.1.1 and A.1.2 for details about the domains. We used all 20 problems available for *Zenotravel* and the first 27 problems from *Blocks*.

5.1.1 Number and distribution of data samples

In total, we obtained 31,434,617 unique data samples for *Blocks* and 25,977,816 for *Zenotravel*. Each data sample is a pair $\langle s^P, h^*(s^P)\rangle$.

Our ad-hoc solvers don't guarantee optimality so the target is actually just a close estimate of h^* as described in section 3.1.2. We will still refer to that number as to h^* throughout this chapter.

Figure 5.1 shows how many samples come from individual problems in the *blocks* domain. Starting with problem 21, we began to use a different setting of our sampling procedure to compensate for increasing problem size. The original technique would no longer finish in reasonable time on larger problems.

Figure 5.2 shows distribution of h^* among problems in blocks domain.

Figures 5.3 and 5.4 show the same for zenotravel.

Figures 5.5 and 5.6 show distribution of samples by their target, i.e. goal distance.

5.2 Heuristics' Accuracy and Adjustments

We used the collected data to analyze accuracy of heuristics, find their weaknesses - if any - any discover possible improvements. Experiments follow the reasoning made in section 2.3. We conducted experiments with two heuristics - *GoalCount* (h_{GC}) and *FastForward* (h_{FF}) on the two domains for which we have the data: *blocks* and *zenotravel*. See section 1.1.5 for details about the heuristics.

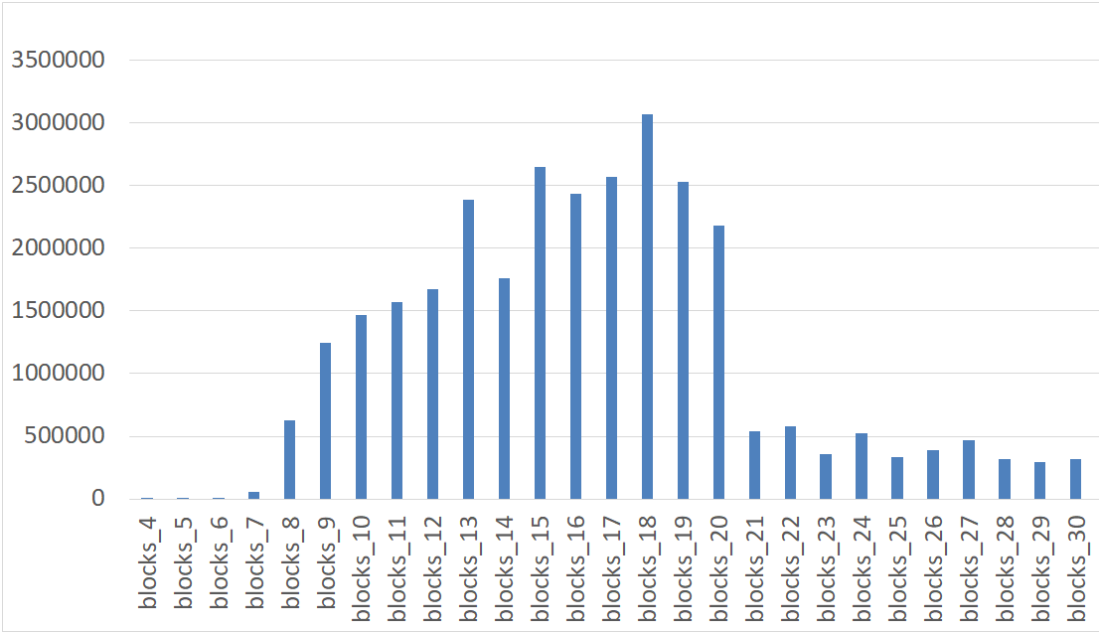

Figure 5.1: Count of samples from individual problems in *blocks* domain.

The experiments were designed as follows: For every state in our data set, we compute value of the two heuristics. We already have the h^* values so this gives us a set of tuples $\{\langle s_i, h_{GC}(s_i), h_{FF}(s_i), h^*(s_i)\rangle\}_i$.

All the data are included in the supplementary materials.

5.2.1 Heuristics' accuracy

Figure 5.7 shows correlation between heuristic estimates and real goal-distances for both heuristic and both domain.

The blue area represents the data as a scatter plot. In each subgraphs, the X-axis is goal distance, Y-axis is the heuristic estimate and the red line represents a perfect match. The closer to the red line the more informed the heuristic is. Data points above the red line indicate overestimation. As expected, h_{FF} is much more informed but not admissible.

Figures 5.8 and 5.9 provide more detailed view on h_{FF}. For every $x \in \mathbb{N}$, we find all states s_i such that $h_{FF}(s_i) = x$ and calculate minimum, average and maximum of their true goal-distances. Formally, let $T_x = \{s_i \mid h_{FF}(s_i) = x\}$, $min^x = \min\{h^*(s_i) \mid s_i \in T_x\}$, $max^x = \max\{h^*(s_i) \mid s_i \in T_x\}$ and $avg^x = avg\{h^*(s_i) \mid s_i \in T_x\}$. For each x on the X-axis, the vertical line represents the interval $[min^x, max^x]$ and the blue dot shows avg^x. Statistics include all collected data from *Blocks* (figure 5.8) and all collected data from *Zenotravel* (figure 5.9).

5.2.2 Performance of adjusted heuristics

Based on the data, we've constructed the three adjusted heuristics h^{min}, h^{avg} and h^{shift} as defined in section 2.3 and tested them on a set of available problems. For every combination of $h \in \{h_{GC}, h_{FF}\}$ and $domain \in \{blocks, zenotravel\}$ we run an A^* search with heuristics h, h^{min}, h^{avg} and h^{shift} on all problems in $domain$.

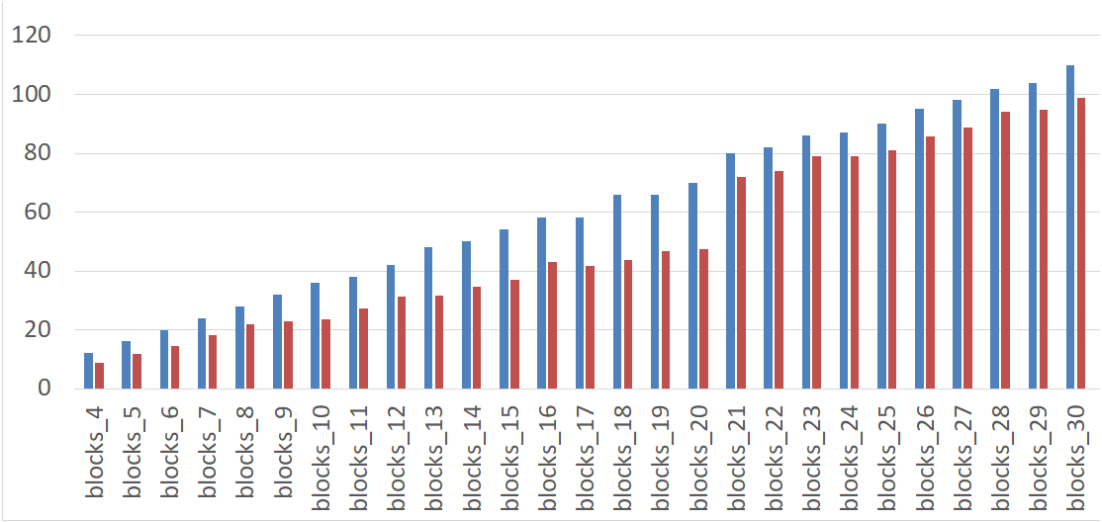

Figure 5.2: Distribution of h^* by problem in *blocks* domain. Red columns show average of h^*, blue ones show maximum. Minimum is always 0.

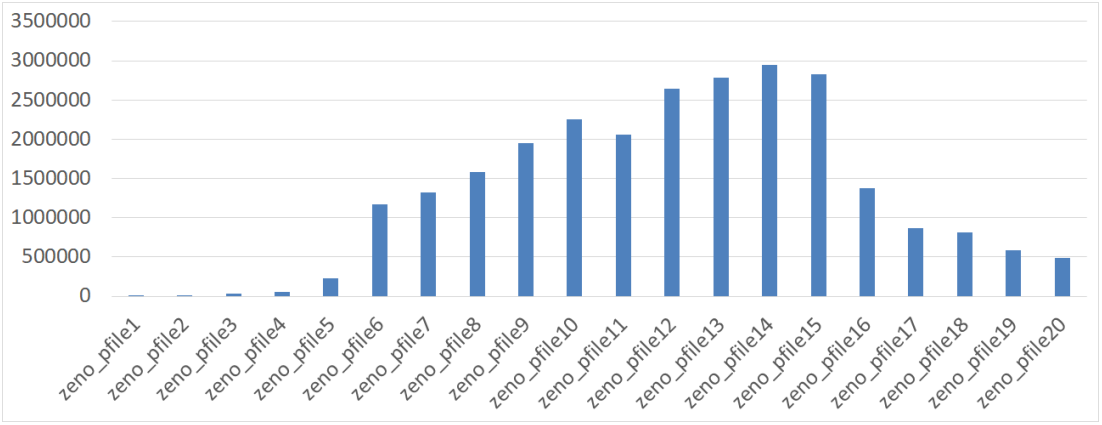

Figure 5.3: Count of samples from individual problems in *zenotravel* domain.

Time was capped at 30 minutes per problem and there was a memory limit of 5 million nodes per problem (sum of sizes of the open list and the closed list).

When solving a problem P, we use only data from *other* problems - all from the domain except P - to construct the adjusted heuristics. This should resemble our use case scenario where we are interested in transferring the knowledge across problems.

Figure 5.10 shows results for *blocks* and figure 5.11 for *zenotravel*. We measure *TotalNodes* - sum of number of expanded nodes and the number of nodes present in the open list at the end of the search, *SolutionCost* which in this case is the number of actions in the plan, number of problems solved and search time. Blank spaces in *SolutionCost* mean that the problem was not solved within the time or memory limit. We've only included problems that were solved by at least one of the heuristics.

GoalCount is an admissible but very week heuristic hence the adjusted heuristics outperform it in all criteria except plan length on all problems in *blocks*. The heuristic is admissible, so the *min* adjustment and the *shift* adjustment will yield the same heuristic, i.e. $\forall s : h^{min}_{GC}(s) = h^{shift}_{GC}(s)$.

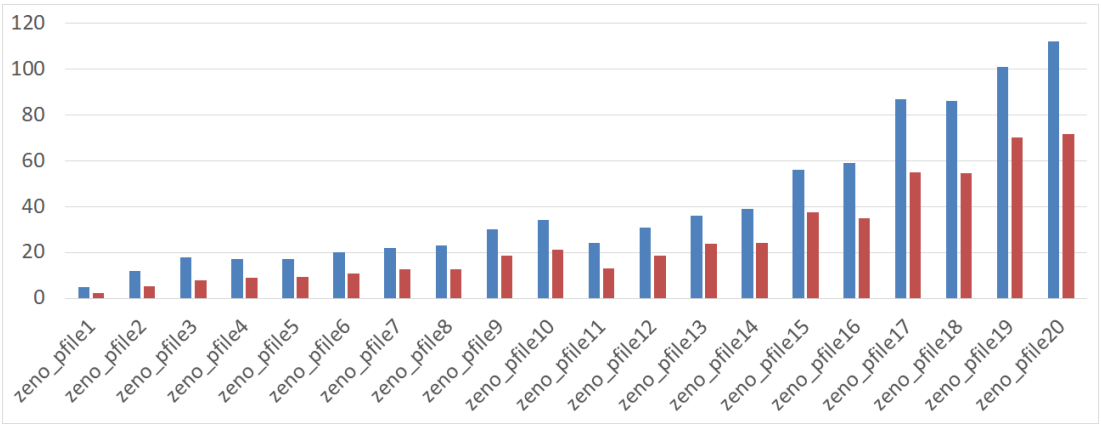

Figure 5.4: Distribution of h^* by problem in *zenotravel* domain. Red columns show average of h^*, blue ones show maximum. Minimum is always 0.

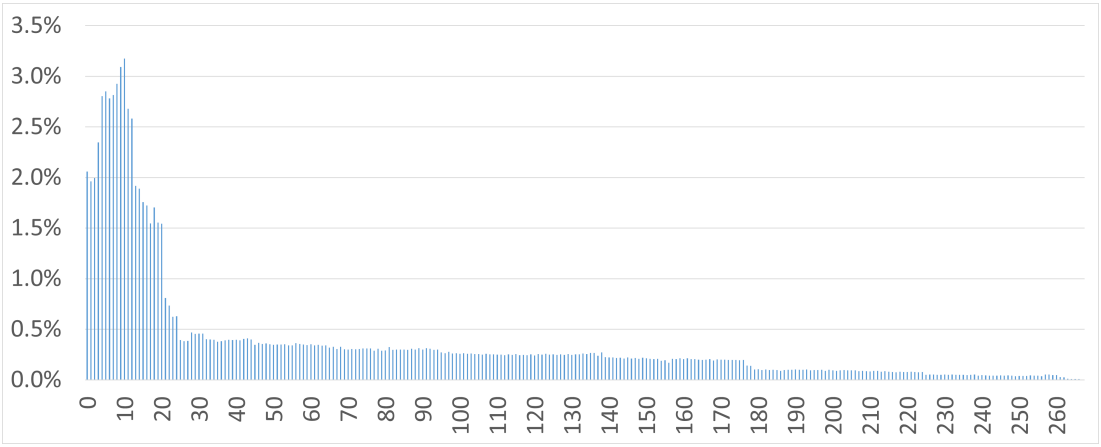

Figure 5.5: Count of samples by their goal-distances. X-axis: goal distance, Y-axis: relative number of samples in *blocks* domain. (All problems combined.)

In *blocks*, the h_{GC} heuristic systematically and unnecessarily underestimates most states. The heuristic counts the number of blocks that are misplaced. Moving a block, however, requires two actions: *lift* and *put-down* hence

$$\forall s : h_{GC}(s) = k \Rightarrow h^*(s) \geq 2k - 1$$

We subtract 1 because one of the k blocks could already by *lifted* in the state. When using the estimate $2k-1$ instead of k, the heuristic is much more informed and still admissible. This is exactly what h^{min}_{GC} and h^{shift}_{GC} are doing.

In *zenotravel* domain, the h_{GC} heuristic counts the number of passengers that are not yet at their destinations. For every k there exists a state s such that $h_{GC}(s) = k = h^*(s)$. The state looks as follows: there are k passengers travelling to the same destination, they are boarded in the plane that is already located at their destination. All other passengers have already disembarked at their respective destinations. In this state, performing the *disembark* action k-times will lead to goal.

Due to this, the heuristic estimate is actually tight and can't be improved without loosing admissibility. Hence in this case $h^{min}_{GC} = h^{shift}_{GC} = h_{GC}$.

h^{avg}_{GC} on the other hand is more informed that the others, solves more problems

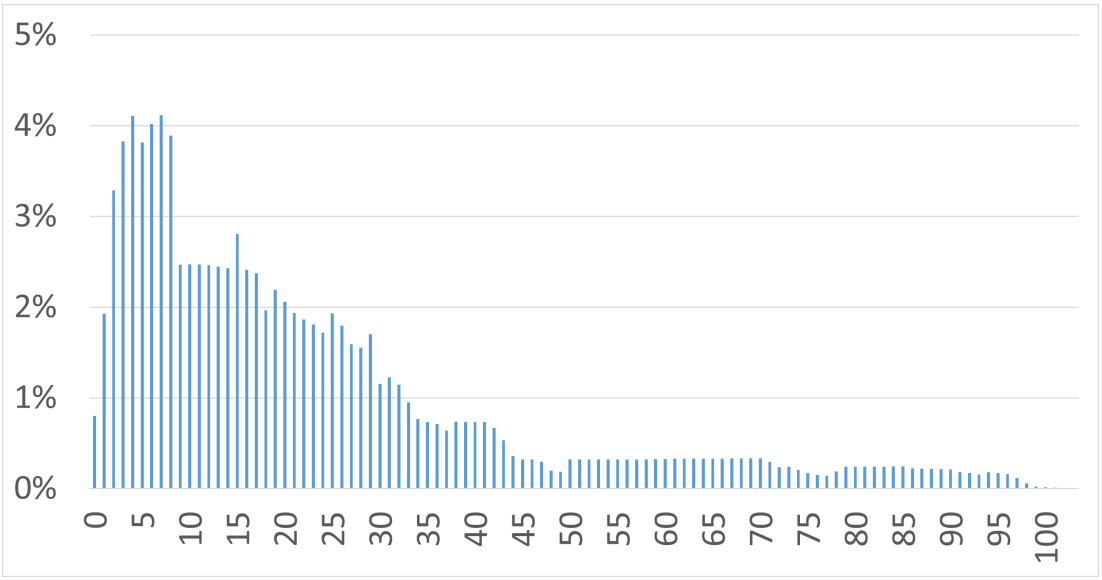

Figure 5.6: Count of samples by their goal-distances. X-axis: goal distance, Y-axis: relative number of samples in *zenotravel* domain. (All problems combined.)

and requires less time and memory. It is not admissible hence it doesn't guarantee optimality of solutions.

The *FF* heuristic is much more sophisticated and it is therefore unlikely that these simple adjustments will significantly improve it. h_{FF}^{Shift} seems to be working exactly like h_{FF} which suggests that there are no obvious errors in h_{FF} that could be repaired by shifting, or at least they don't occur on either of the two domains. None of the Avg and Min adjustments outperform h_{FF} in all criteria. h_{FF}^{Min} is admissible unlike h_{FF} hence can improve the solution quality and provides optimality guarantee. h_{FF}^{Avg} can sometimes be faster, especially in the *blocks* domain at the cost of increasing plan length.

5.2.3 Summary

The *min* and *shift* adjustments are able to automatically identify and repair the underestimation that occurs with h_{GC} on *blocks*. h_{GC}^{avg} outperforms h_{GC} in search time, memory consumption as well as number of problems solved at the cost of loosing optimality.

Improvements over h_{FF} are negligible but we should point out that developing h_{FF} took several years and many man-days of work of top-tier researchers while the adjusted heuristic can be constructed automatically within minutes, assuming that we already have the data.

The proposed modification are applicable to any existing heuristic and can be used to adjust the informedness vs. admissibility tradeoff of the heuristic.

5.3 Expressive Power of Features

We experimentally measure the expressive power of our features extractor as discussed in section 4.3.3 using the data we've collected.

Table 5.1 shows length of feature vectors $F_\alpha(\cdot)$ based on parameter α.

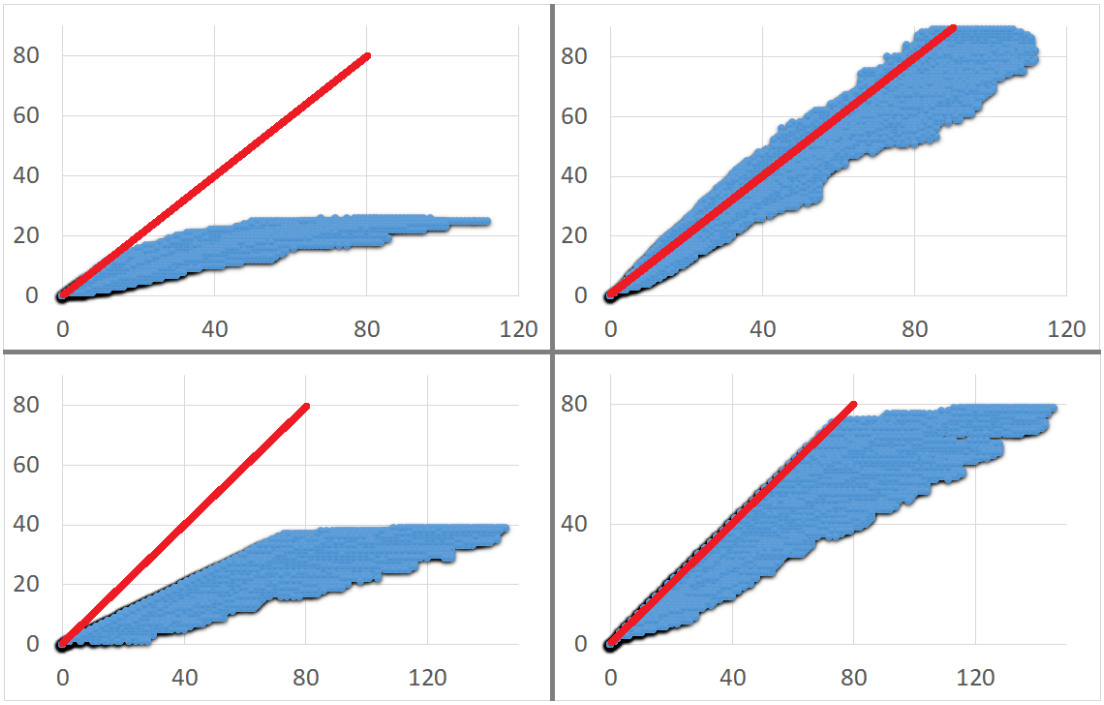

Figure 5.7: Correlation between heuristic estimates and real goal-distances. Top section: zenotravel, bottom section: blocks. Left hand side: h_{GC}, right hand side: h_{FF}.

Size of subgraphs (α)	$\lvert F_\alpha \rvert$ in Zenotravel	$\lvert F_\alpha \rvert$ in Blocks
2	12	11
3	36	26
4	107	57
5	343	139
6	1140	

Table 5.1: Length of feature vectors produced by our method for various values of α on the two domains.

As explained in section 4.3.3, states with the same features are indistinguishable to the ML model and hence will be assigned the same output - the same heuristic estimate. It is therefore crucial that these indistinguishable states also have similar h^*, i.e. that the *spread* of h^* amongst the indistinguishable states is low.

To measure the expressive power of features, we use the following approach. First we group together states with the same feature vectors, i.e., we construct Δ-sets. Then we take a look at distribution of values $h^*(\cdot)$ of states in each such group, similar to the analysis in figures 5.8 and 5.9. In every Δ-set, we calculate minimum, maximum and standard deviation of values $h^*(\cdot)$ of corresponding states and we calculate *range* as *range* = *maximum* - *minimum*.

Figure 5.12 shows distribution of *ranges* for two values of α. On the X-axis there is range and height of the column represent the amount of states that belong to Δ-sets with that range. Specifically: each state belongs to exactly one Δ-set, and each Δ-set is assigned a range. We can hence group states with the

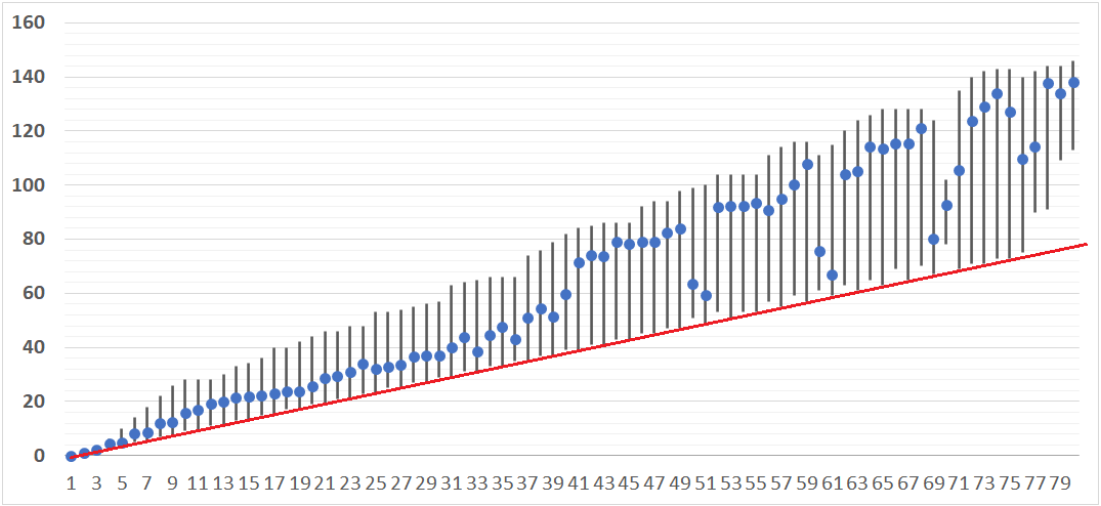

Figure 5.8: Distribution of true goal-distances for each heuristic estimate in *Blocks* domain. X-axis: h_{FF} estimates, Y-axis: minimum, average and maximum of h^* of states having the corresponding estimate. Data below the red line indicate over-estimation of the heuristic.

same range together and plot relative size of each such group. Blue columns corresponds to features computed with $\alpha = 2$ and orange ones to value $\alpha = 4$. The lower means better as it indicates lower spread and more coherent groups.

For example, the graph shows that for $\alpha = 2$ the category 54 is populated, meaning that there are states s_1, s_2 with the same features vectors whose $|h^*(s_1) - h^*(s_2)| = 54$. These states will be indistinguishable by the learned heuristic hence they will be assigned the same estimate. No matter of what that estimate will be, the error of the heuristic will be at least $^{54}/_2$ on at least half of states that belong to the category 54.

On the other hand, for $\alpha = 4$ the highest populated category is 24 and majority of states belong to categories $0 \sim 12$. Error of the learned heuristic will be at most 12 and should be around $^{12}/_2$ or less on these states.

Table 5.2 displays average range with respect to parameter α. For each Δ-set, we calculate the range and then weighted average of those ranges where weights are numbers of states in corresponding Δ-sets. Table 5.3 shows average standard deviations calculated in the same way.

Size of subgraphs (α)	average range in Zenotravel	average range in Blocks
2	17.00	24.74
3	12.67	24.53
4	5.75	10.74
5	2.75	3.64

Table 5.2: Average range of features' groups for various values of α.

As expected, higher values of α lead to smaller and more coherent groups of states that share the same features, i.e. they improve the expressive power of the features extractor.

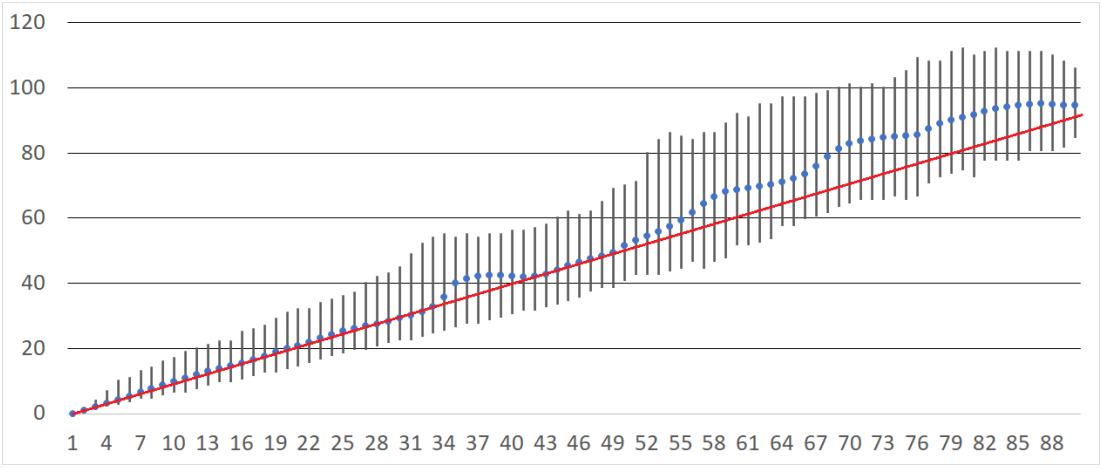

Figure 5.9: Distribution of true goal-distances for each heuristic estimate in *Zenotravel* domain. X-axis: h_{FF} estimates, Y-axis: minimum, average and maximum of h^* of states having the corresponding estimate. Data below the red line indicate over-estimation of the heuristic.

Size of subgraphs (α)	avg. stdDev in Zenotravel	avg. stdDev in Blocks
2	2.39	2.96
3	2.13	2.94
4	1.30	2.31
5	0.83	1.02

Table 5.3: Average standard deviations in features' groups for various values of α.

5.3.1 Correlation between features and targets

We measure correlation of differences between features and differences between targets among states. This helps us answer the question of whether states with similar *targets* also have similar *features*. This property is important as it improves the generalization capabilities of the model and makes the learning task easier in general, especially in the regression setting. See [74] section 6.1 for a more thorough explanation.

Given two samples $t_1 = (s_1, h^*(s_1))$ and $t_2 = (s_2, h^*(s_2))$, we calculate difference between features as $d(f(s_1), f(s_2))$ and difference between targets as $|h^*(s_1) - h^*(s_2)|$. We calculate these for many pairs t_i, t_j and measure correlation coefficient between difference in features and difference in targets. Features are points in $\mathbb{R}^k$. We use three different distance metrics to calculate features-differences: Euclidean distance (L_2-norm), MAE (L_1-norm), and Chebyshev distance (L_∞-norm). The results are very similar for all these metrics.

Ideally, we would like to include all pairs t_i, t_j in the calculation but since we have more than 10^6 samples, the number of pair is too great so instead we sample 10^7 pairs uniformly randomly and use these as a representative sample.

Figure 5.13 shows a scatter plot of the data. Each point represents a pair of training samples. X-coordinate shows Euclidean distance between their features while Y-coordinate represents difference in their targets for $\alpha = 2$ in *Blocks* domain. The graph alone indicates that the two are highly correlated with exception

	NodesTotal				SolutionCost				SearchTime (s)			
Problem	**GC**	**GC+avg**	**GC+min**	**GC+shift**	**GC**	**GC+avg**	**GC+min**	**GC+shift**	**GC**	**GC+avg**	**GC+min**	**GC+shift**
probBLOCKS-4-0.sas	54	48	27	27	6	10	6	6	0	0	0	0
probBLOCKS-5-0.sas	241	46	46	46	12	12	12	12	0	0	0	0
probBLOCKS-6-0.sas	304	79	48	48	12	12	12	12	0	0	0	0
probBLOCKS-7-0.sas	6040	233	602	602	20	22	20	20	0	0	0	0
probBLOCKS-8-0.sas	73194	2114	1680	1680	18	22	18	18	1	0	0	0
probBLOCKS-9-0.sas	4E+06	1E+06	264398	264398	30	34	30	30	83	22	4	4
probBLOCKS-10-0.sas	5E+06	4E+06	3E+06	3E+06		36	34	34	98	71	56	56
probBLOCKS-11-0.sas	5E+06	5E+06	2E+06	2E+06			32	32	101	97	30	29
probBLOCKS-12-0.sas	5E+06	5E+06	4E+06	4E+06			34	34	105	96	67	65
Problem	**FF**	**FF+avg**	**FF+min**	**FF+shift**	**FF**	**FF+avg**	**FF+min**	**FF+shift**	**FF**	**FF+avg**	**FF+min**	**FF+shift**
probBLOCKS-4-0.sas	20	18	18	20	6	6	6	6	0	0	0	0
probBLOCKS-5-0.sas	45	29	40	45	12	12	12	12	0	0	0	0
probBLOCKS-6-0.sas	38	36	33	38	12	12	12	12	0	0	0	0
probBLOCKS-7-0.sas	215	89	226	215	20	22	20	20	0	0	0	0
probBLOCKS-8-0.sas	927	572	650	927	18	22	18	18	0	0	0	0
probBLOCKS-9-0.sas	35585	104938	38200	35585	30	30	30	30	21	58	20	19
probBLOCKS-10-0.sas	1E+06	820863	685206	1E+06	34	38	34	34	885	602	472	817
probBLOCKS-11-0.sas	128839	100648	209636	128839	32	34	32	32	123	88	183	114
probBLOCKS-12-0.sas	137680	53302	221442	137680	34	36	34	34	175	65	259	167
probBLOCKS-14-0.sas	615017	902475	398023	615017	38		38	38	1283	1800	768	1214

Figure 5.10: Performance of h_{FF}, h_{GC} and their modified versions in *blocks*.

	NodesTotal				SolutionCost				SearchTime (s)			
Problem	**GC**	**GC+avg**	**GC+min**	**GC+shift**	**GC**	**GC+avg**	**GC+min**	**GC+shift**	**GC**	**GC+avg**	**GC+min**	**GC+shift**
pfile1.sas	5	5	5	5	1	1	1	1	0	0	0	0
pfile2.sas	88	107	88	88	6	6	6	6	0	0	0	0
pfile3.sas	1377	643	1377	1377	6	6	6	6	0	0	0	0
pfile4.sas	4042	4426	4042	4042	8	8	8	8	0	0	0	0
pfile5.sas	15802	4275	15802	15802	11	13	11	11	0	0	0	0
pfile6.sas	97502	22345	97502	97502	11	13	11	11	1	0	2	1
pfile7.sas	172953	46596	172953	172953	15	16	15	15	2	1	3	2
pfile8.sas	596687	35040	596687	596687	11	12	11	11	9	0	10	9
pfile9.sas	5E+06	1E+06	5E+06	5E+06		21			114	20	114	115
pfile10.sas	5E+06	4E+06	5E+06	5E+06		23			105	59	97	104
pfile11.sas	5E+06	268131	5E+06	5E+06		14			117	4	109	119
Problem	**FF**	**FF+avg**	**FF+min**	**FF+shift**	**FF**	**FF+avg**	**FF+min**	**FF+shift**	**FF**	**FF+avg**	**FF+min**	**FF+shift**
pfile1.sas	5	5	5	5	1	1	1	1	0	0	0	0
pfile2.sas	41	41	43	41	6	6	6	6	0	0	0	0
pfile3.sas	97	97	293	97	6	6	6	6	0	0	0	0
pfile4.sas	123	123	412	123	8	8	8	8	0	0	0	0
pfile5.sas	122	122	1378	122	11	11	11	11	0	0	0	0
pfile6.sas	185	231	4010	185	12	12	11	12	0	0	1	0
pfile7.sas	2870	2826	25968	2870	15	15	15	15	1	1	8	1
pfile8.sas	594	594	8356	594	12	12	11	12	0	0	6	0
pfile9.sas	2146	5185	3E+06	2146	21	21		21	1	3	1800	1
pfile10.sas	9430	39215	2E+06	9430	23	23		23	8	36	1800	8
pfile11.sas	4157	9863	780032	4157	14	14	14	14	6	15	968	6
pfile12.sas	29221	63698	1E+06	29221	21	21		21	43	96	1800	44
pfile13.sas	130732	172523	899246	130732	26	26		26	226	335	1800	225
pfile14.sas	39666	247819	193657	39666	30			30	304	1800	1800	305

Figure 5.11: Performance of h_{FF}, h_{GC} and their modified versions in *zenotravel*.

of the lower-left region that seems more erratic.

Figure 5.14 shows correlation coefficients for various values of α on the two domains. The correlation is high in all cases which indicates that our features are well chosen. The correlation coefficient seems to go down slightly for $\alpha > 3$. We believe that this is caused by the *curse of dimensionality*. As α increases, dimensionality of the features-space increases as well which means that features

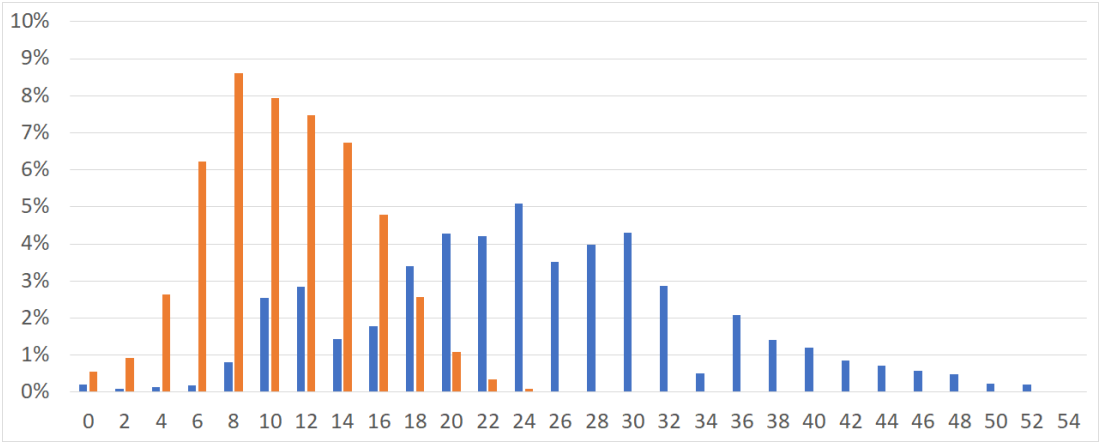

Figure 5.12: Distribution of *ranges* of Δ-sets in *blocks*. Blue graph corresponds to $\alpha = 2$, the orange one to $\alpha = 4$. Calculated over all collected data from *Blocks* domain.

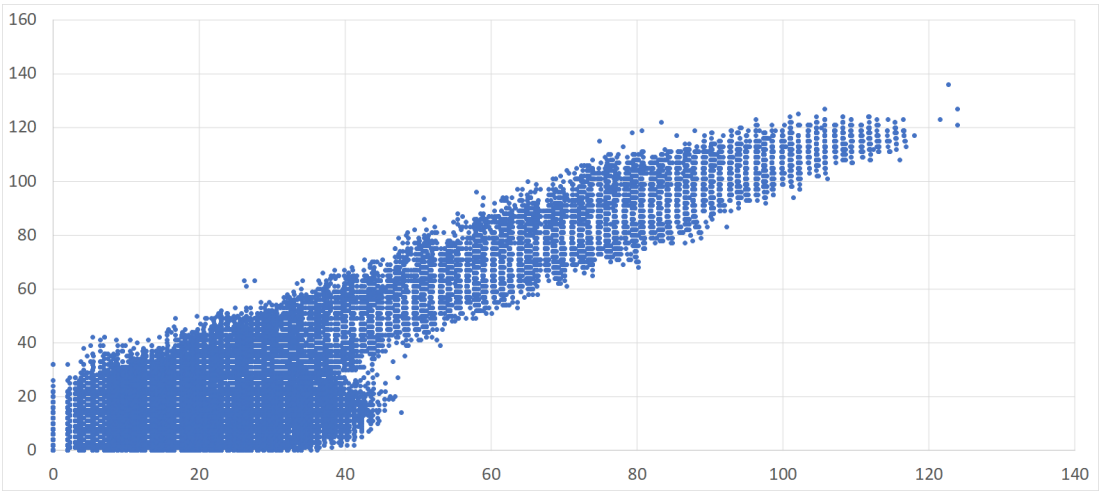

Figure 5.13: Scatter plot of features' differences against targets' differences.

are more likely to be far from each other even if the relative number of indices on which they differ stays the same.

The raw data are included in the supplementary materials.

5.4 Performance of Learned Heuristics

In this section, we present our main experimental results. We compare heuristics created by our HL framework with a popular domain independent heuristic on a set of planning problems. We use *FastForward* heuristic ([48]) as a baseline. The heuristic has been around for quite some time now and it is still often used as a baseline in experiments, like in [49] for example.

We experiment with two domain: *Zenotravel* and *Blocks* and we run the HL process separately on each domain.

Correlation		Domain	
		blocks	zeno
α	2	78.8%	89.1%
	3	87.5%	90.1%
	4	79.7%	87.1%
	5	71.3%	84.1%

Figure 5.14: Correlation coefficients between features-distance and targets-distance for various values of α.

5.4.1 Generalization capabilities

The intended use-case scenario, as described in section 3, looks as follows. We are given s set of problems all from the same domain that can be used for training. Afterwards the trained model should be able to quickly solve new, previously unseen problems from the same domain. To emulate this scenario, we use the *one-leave-out* strategy ([35]).

We've collected training data from problems $P_1, \ldots, P_n$. For each problem P_i, we train the network using data from all other problems except P_i. Then we use the learned heuristic to solve P_i.

This way we have to train many different models instead of just one for the whole domain, but it allows us to verify that the knowledge is transferable across different problem instance.

5.4.2 Choice of hyper-parameters

We use a NN with 5 fully-connected hidden layers with sizes of $(256, 512, 128, 64, 32)$ neurons respectively, and three *DropOut* layers. We used *ReLU* activation function, *Xavier* weight initialization and *Adam* as the training algorithm (see e.g. [56, 35]). The input layer size is the same as size of the feature vectors and the last layer contains a single neuron with a linear activation to compute the output. Figure 5.15 visualizes the architecture.

The network is large enough to create efficient representation of the data and drop-out layers together with large number of data samples prevent overfitting. The training was terminated when 1000 epochs passed without improving the test error. The model with the lowest test error over-all was used in the deployment.

We experimented with values of parameter α (size of subgraphs) from 2 to 4. For values larger that 4, computing features of states is too costly and the heuristic is not competitive on the domains we work with. We experimented with two loss functions: *MSE* and *LogMSE* - see section 3.3.1 for definitions. We also tried to include the value of h_{FF} among the features of states. I.e., we first trained the network having only the graph-based features as its inputs and then another network that used both graph-based features and h_{FF} value of the state as its inputs. We conducted experiments for all combinations of these parameters: $\alpha \in \{2, 3, 4\}$, $lossFunction \in \{MSE, LogMSE\}$, $FFasFeature \in \{true, false\}$. This gives us 12 different neural net-based heuristics for each problem.

The experiments were performed on *HP Z640 Workstation* with 64 GB of RAM, 8 CPU cores. We trained the models using a CUDA-capable graphics card *NVIDIA Quadro M5000* with 8 GB of memory and 2048 GPU cores.

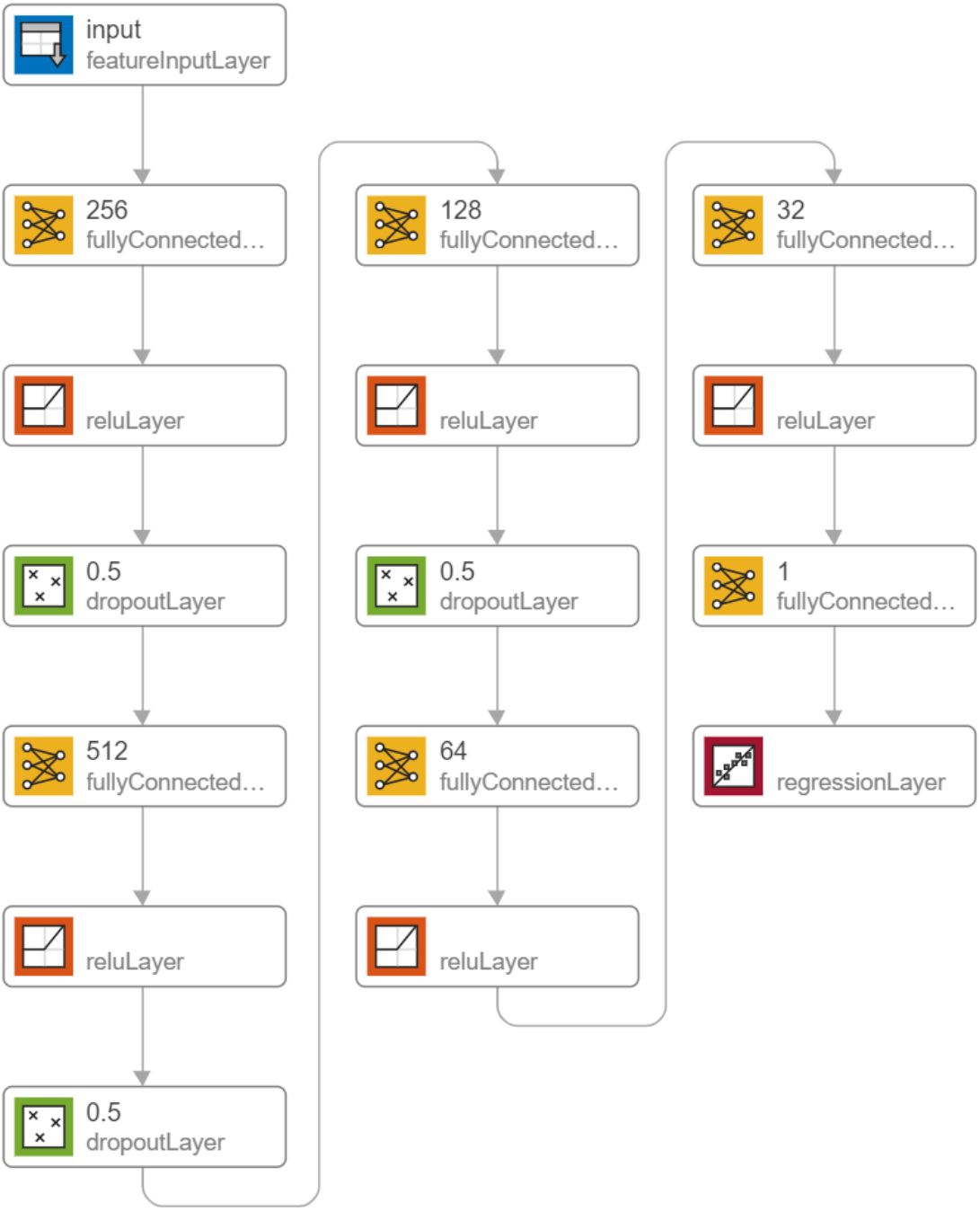

Figure 5.15: Architecture of the network.

The training took about 120 hours in total for *zenotravel* and about 175 hours for *blocks*. We used the *BrightWire*[1] NN framework to train the models.

5.4.3 Training results

This section presents results of training the networks. We present the results for the best performing variant, which is $\alpha = 4$, *lossFunction* = *LogMSE*, *FFasFeature* = *true*. A comparison of performance of networks trained by *MSE* and *LogMSE* is provided in section 5.5.

[1] http://www.jackdermody.net/brightwire

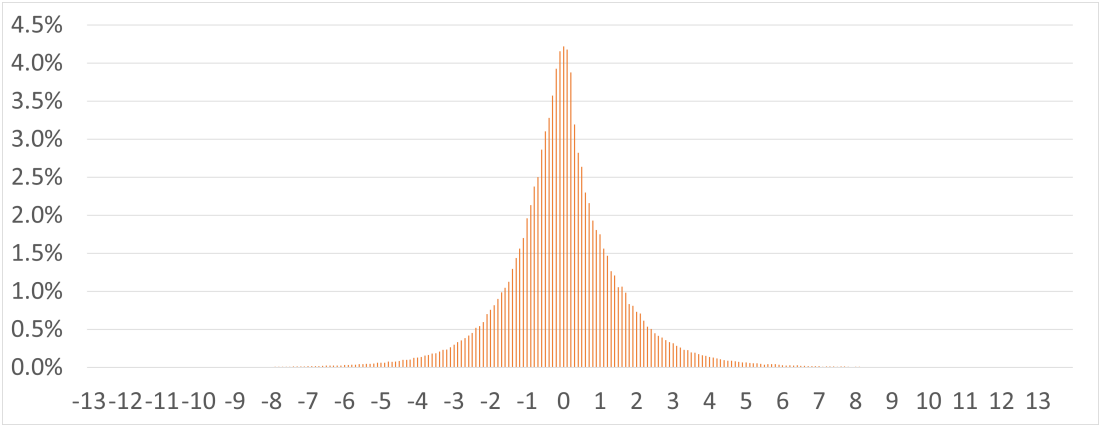

Figure 5.16: Accuracy of fit in *zenotravel*. X-axis: difference between *target* and *output*, Y-axis: percentage of samples in each category

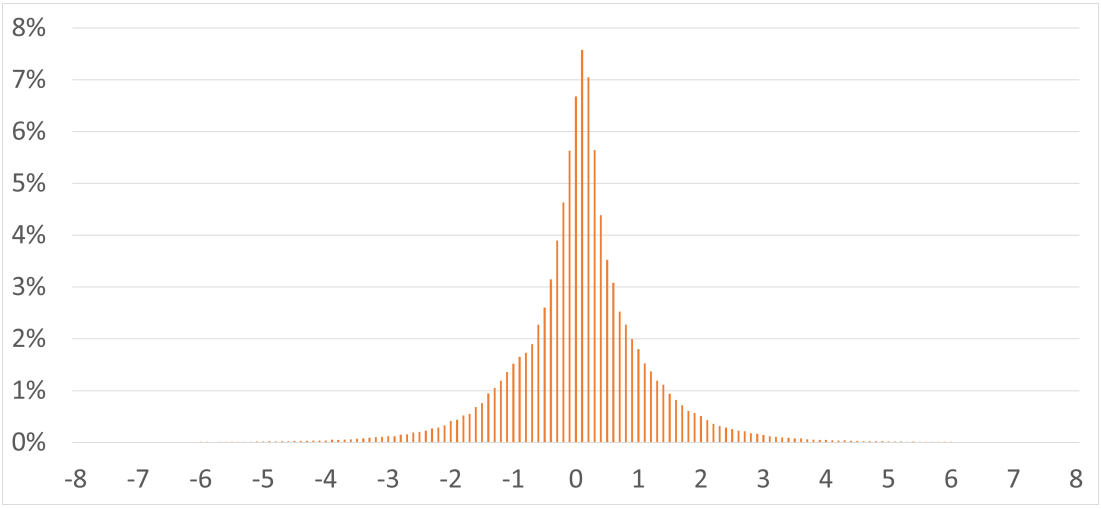

Figure 5.17: Accuracy of fit in *blocks*. X-axis: difference between *target* and *output*, Y-axis: percentage of samples in each category

Figures 5.16 and 5.17 show accuracy of fit of the models. The average absolute difference (MAE) defined as $MAE = avg_i\{|y_i - \hat{y}_i|\}$ is $MAE = 0.75$ in *blocks* and $MAE = 0.98$ in zenotravel. In blocks, the minimum difference is -21.6 on sample $(y = 141, \hat{y} = 119.4)$ and maximum difference 18.6 on sample $(y = 133, \hat{y} = 151.6)$. In zenotravel, the minimum and maximum differences are -13.9 on $(y = 80, \hat{y} = 94.9)$ and 13.7 on $(y = 94, \hat{y} = 80.3)$. The 80%-quantile of absolute difference is 1.19 in blocks, meaning that absolute difference is less than 1.19 on more than 80% of samples. The same quantile in zenotravel is 1.5.

We also measure *ratio* computed as $ratio_i = \max(y_i + 1/\hat{y}_i + 1, \hat{y}_1 + 1/y_i + 1)$ for each individual sample. In blocks, the average and maximum ratio is: average $= 1.05$, max $= 5.62$ on sample $(y = 3, \hat{y} = 21.5)$. In zenotravel, these values are: average $= 1.07$, max $= 3.36$ on sample $(y = 3, \hat{y} = 0.2)$. (Minimum ratio is always 1.)

Figures 5.18 and 5.19 present differences for various targets. We can see that the models fit the data quite well in both domains. Better accuracy is achieved on *blocks* domain. This seems to be domain specific.

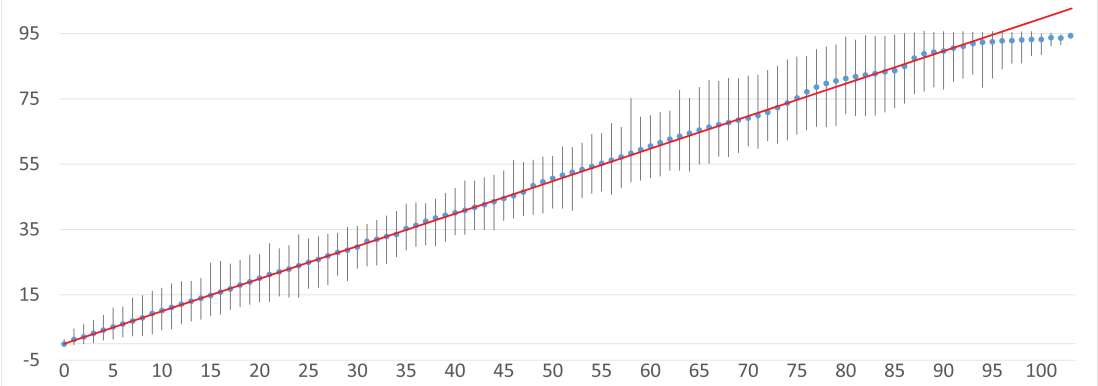

Figure 5.18: Output of the model by target in *zenotravel*. X-axis: target, Y-axis: minimum, maximum and average of outputs on samples with that target. Red line represents perfect fit. Data above the red line indicate overestimation of the learned heuristic.

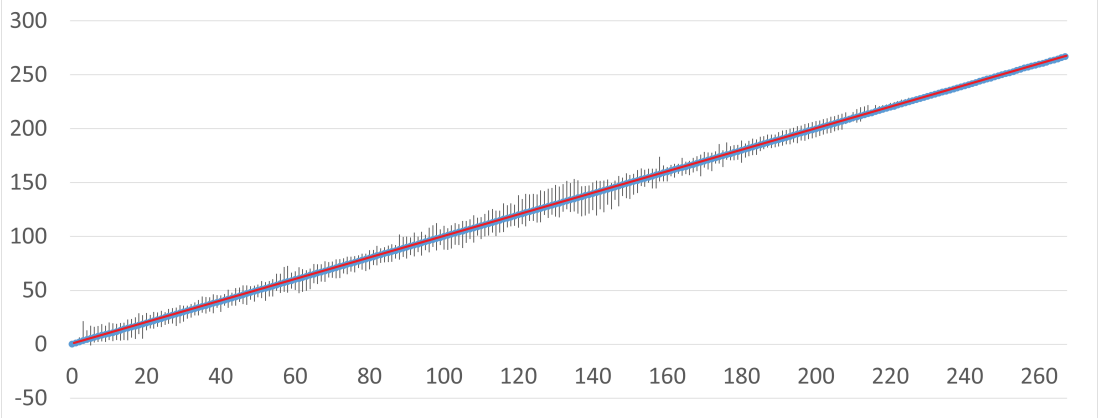

Figure 5.19: Output of the model by target in *blocks*. X-axis: target, Y-axis: minimum, maximum and average of outputs on samples with that target. Red line represents perfect fit. Data above the red line indicate overestimation of the learned heuristic.

5.4.4 Performance evaluation

We used all 13 heuristics (12 NN-based + h_{FF}) to solve each of the problems. Search time was capped at 30 minutes per problem instance. None of the heuristics is admissible so they don't guarantee finding optimal plans. We compared performance of heuristics using the *IPC-Score*. Given a search problem P, a minimization criterion R (e.g. length of the plan) and algorithms $A_1, A_2, \ldots, A_k$, the IPC-Score of A_i on problem P is computed as follows:

$$IPC_R(A_i, P) = \begin{cases} 0, & \text{if } A_i \text{ didn't solve } P \\ \frac{R^*}{R_i}, & \text{otherwise} \end{cases}$$

where R_i is value of the criterion for the i-th algorithm and $R^* = \min_i\{R_i\}$. For every problem P, $IPC_R(A_i, P) \in [0, 1]$ and higher means better. We can then sum up the IPC-Score over several problem instances to get accumulated results. The IPC-Score takes into account both number of problems solved as well as quality of solutions found. We monitor four criteria:

- total number of problems solved
- IPC-Score of time
- IPC-Score of plan length
- IPC-Score of number of expansions

5.4.5 Results

Figure 5.20 shows aggregated results for the two domains: number of problems solved and IPC scores. Tables with detailed results follow in figures 5.21, 5.22, 5.23, 5.24, 5.25, 5.26. We present results of *plan length*, *number of expanded nodes* and *total runtime* for every problem and each heuristic. We only include problems solved by at least one method. There are 13 heuristics in the comparison: *FastForward* heuristic as the baseline and 12 variants of learned heuristics described above. Raw data can be found in the supplementary materials.

			IPC Score - Time		IPC Score - Plan length		IPC Score - Expansions		Solved problems	
heuristic	α	FF used	blocks	zenotravel	blocks	zenotravel	blocks	zenotravel	blocks (27 total)	zenotravel (20 total)
NN heuristic (MSE)	2	FALSE	3.02	2.00	4.86	4.00	0.35	0.74	5	4
		TRUE	6.39	3.09	21.63	6.58	5.47	1.86	22	7
	3	FALSE	3.69	2.02	7.00	6.06	1.59	0.21	7	7
		TRUE	9.59	2.24	24.58	7.76	8.89	1.92	25	8
	4	FALSE	6.66	2.17	21.73	8.00	7.25	1.48	22	8
		TRUE	7.59	4.19	19.82	10.08	7.75	4.08	20	10
NN heuristic (LogMSE)	2	FALSE	5.31	2.01	9.98	5.92	1.79	0.74	10	6
		TRUE	14.99	4.16	24.75	9.30	10.89	3.96	25	9
	3	FALSE	8.43	2.37	11.98	10.52	4.82	3.26	12	11
		TRUE	**21.57**	5.31	**25.94**	11.08	17.77	5.53	**26**	11
	4	FALSE	18.67	4.46	**25.94**	12.08	20.68	4.77	**26**	12
		TRUE	19.57	8.49	**25.94**	**15.08**	**22.61**	**10.10**	**26**	**16**
FF heuristic			5.03	**13.00**	5.06	10.72	0.32	9.37	8	13

Figure 5.20: Aggregated results of experiments.

We can see that networks trained by LogMSE are superior to the ones trained by MSE in all criteria on both domains. We will investigate this more in the following section.

As expected, higher values of α lead to a more accurate heuristic: the number of expanded nodes as well as plan length are better. The difference is apparent especially for values 2 and 3. Using value $\alpha = 4$ still helps but computing features in this case is slower and so A* expands less nodes per second and the overall results are not that much better than for $\alpha = 3$.

Adding h_{FF} as feature has a mixed effect. It is very helpful on *blocks* domain when $\alpha \in \{2, 3\}$, but not much helpful when $\alpha = 4$. See figure 5.27. This indicates that subgraphs of size 2 and 3 cannot capture useful knowledge about a blocks problem hence the network rely on h_{FF} as the source of information. Subgraphs of size 4 seem to be able to provide the required knowledge already and adding h_{FF} doesn't help anymore. This agrees with results in table 5.2 which shows that average *range* of Δ-sets doesn't change much between $\alpha = 2$ and $\alpha = 3$ but there is a large drop for $\alpha = 4$.

The need to evaluate h_{FF} on every expansion slows down the search so the performance is worse than not using h_{FF} at all in this case. In general, adding h_{FF} improves accuracy of the NN so the resulting heuristic is more informed

Expanded nodes	Fast Forward	FF heuristic as feature: FALSE						FF heuristic as feature: TRUE					
		NN heuristic (MSE)			NN heuristic (LogMSE)			NN heuristic (MSE)			NN heuristic (LogMSE)		
problem	heuristic	2	3	4	2	3	4	2	3	4	2	3	4
probBLOCKS-4-0	69	23	40	10	82	30	10	21	10	7	18	10	10
probBLOCKS-5-0	112	689	134	37	556	244	27	46	18	21	136	47	22
probBLOCKS-6-0	539	1343	659	37	1219	385	21	343	24	40	476	25	29
probBLOCKS-7-0	3018	27194	5141	161	19522	7943	30	5040	1094	56	5850	608	52
probBLOCKS-8-0	4984	98079	8973	241	20275	17897	165	1604	2345	158	1083	933	49
probBLOCKS-9-0	603221			435		180505	314	9651	603	410	3472	339	98
probBLOCKS-10-0	149202			922			35	69249	1603	1083	4746	795	35
probBLOCKS-11-0				1172			307	4007	1182	682	1832	411	223
probBLOCKS-12-0	81515			505			114	350	249	209	146	212	107
probBLOCKS-13-0				900			328	12393	1165	1452	816	274	96
probBLOCKS-14-0				245			126	864	360	534	488	146	122
probBLOCKS-15-0				1514			64	4345	850	1206	1897	108	64
probBLOCKS-16-0				9927			721		728	1584		672	717
probBLOCKS-17-0				6360			287	5291	1650	641	2607	111	287
probBLOCKS-18-0							4382	1261			9256	1127	4176
probBLOCKS-19-0				5658			177	8610	3680	5921	810	562	177
probBLOCKS-20-0				718			793	3521	899	1558	692	743	793
probBLOCKS-21-0				3050	44438		457		20956		3375	529	457
probBLOCKS-22-0							5135	19734	7968		4613	4401	5135
probBLOCKS-23-0							2624	20017	5709	8810	2555	2624	2624
probBLOCKS-24-0						23860	10423	8490	15302		10255	10420	10423
probBLOCKS-25-0				2483		3847	1428		41652		1428	1428	1428
probBLOCKS-27-0			2248	7915	1041	542	562	3204	2864	1119	562	562	562
probBLOCKS-28-0				14280	23837	2513	2523	5374	12209	11157	2523	2523	2523
probBLOCKS-29-0			3702	4981	359340	5084	5092		23875		5092	5092	5092
probBLOCKS-30-0				7743	7715	7743	7743	7743	7743	7743	7743	7743	7743

Figure 5.21: Results of experiments - number of expanded nodes by each heuristic on each problem on *blocks*. Only problems solved by at least one method are included.

which improves both number of expansions and plan length. Due to the slowdown, though, adding h_{FF} doesn't always improve runtime and the number of problems solved.

Figure 5.28 shows the same for Zenotravel. Here adding h_{FF} as feature is beneficial for all values of α and for both loss functions.

Among the neural-based heuristics, the setting with $\alpha = 4$, h_{FF} added and *LogMSE* performs best. When we compare it with the h_{FF}, we see that our method vastly outperforms the baseline on *blocks* where it solved 26 out of 27 problems while h_{FF} can only solve 8 problems. Even on problems solved by both methods, the NN heuristic finds shorter plans and expands less nodes which is apparent in figures 5.23 and 5.21.

On the *zenotravel* domain, our method outperforms h_{FF} in all criteria except *Time*. As h_{FF} can find suboptimal plans very quickly in *zenotravel*, it is difficult to achieve better score even though our method solved more problems within the time limit. On problems solved by multiple methods, h_{FF} finds longer plans than the best variant of learned heuristic, as figure 5.26 illustrates.

5.5 MSE versus LogMSE

The experiments showed that heuristics trained by *MSE* don't perform well during the search. Figures 5.29 and 5.30 show comparison of number of problems solved.

Runtime (sec.)	Fast Forward	FF heuristic as feature: FALSE						FF heuristic as feature: TRUE					
		NN heuristic (MSE)			NN heuristic (LogMSE)			NN heuristic (MSE)			NN heuristic (LogMSE)		
problem	heuristic	2	3	4	2	3	4	2	3	4	2	3	4
probBLOCKS-4-0	0	0	0	0	0	0	0	0	0	0	0	0	0
probBLOCKS-5-0	0	1	0	0	1	0	0	0	0	0	0	0	0
probBLOCKS-6-0	0	3	1	0	2	1	0	1	0	0	1	0	0
probBLOCKS-7-0	0	48	13	1	24	12	0	13	4	0	15	2	0
probBLOCKS-8-0	1	213	29	2	39	33	1	6	10	1	4	4	0
probBLOCKS-9-0	222			4		379	3	49	4	4	17	2	1
probBLOCKS-10-0	78			13			0	377	14	16	30	7	0
probBLOCKS-11-0				16			3	25	9	10	10	3	2
probBLOCKS-12-0	59			7			1	2	2	3	1	1	1
probBLOCKS-13-0				23			5	151	16	41	7	3	1
probBLOCKS-14-0				5			1	7	3	12	3	1	1
probBLOCKS-15-0				58			1	60	12	43	15	1	1
probBLOCKS-16-0				525			14		12	66		9	14
probBLOCKS-17-0				405			4	73	29	30	25	1	4
probBLOCKS-18-0							77	20			103	15	76
probBLOCKS-19-0				505			3	153	81	483	8	6	3
probBLOCKS-20-0				56			11	40	13	102	4	6	11
probBLOCKS-21-0				264	239		9		457		38	6	9
probBLOCKS-22-0							84	400	158		44	51	86
probBLOCKS-23-0							40	515	117	864	21	28	41
probBLOCKS-24-0						88	141	167	326		69	88	143
probBLOCKS-25-0				192		15	19		813		11	13	19
probBLOCKS-27-0			17	518	2	1	5	51	41	68	2	3	5
probBLOCKS-28-0				873	60	6	21	85	198	690	11	12	21
probBLOCKS-29-0			16	213	681	11	39		223		19	21	37
probBLOCKS-30-0				334	18	20	65	73	89	338	36	38	62

Figure 5.22: Results of experiments - runtime for each heuristic on each problem on *blocks*. Only problems solved by at least one method are included.

In this section, we analyze this phenomenon and we argue that *LogMSE* is much better choice for HL.

Our analysis shows that the poor performance is caused by the distribution of error among samples. (See section 1.2.3 for some background.) Figure 5.31 shows distribution of absolute error between samples based on their targets. The error is larger on states with higher distance-to-go which is expected. The further from a goal the state is the harder it should be to accurately estimate its goal distance.

We can see that the error is distributed relatively evenly among samples with goal distance roughly between 0 and 50. The error is between 1 and 2 on all those states. For example, the model achieves error of ± 1.5 on states 50 steps from goal, but makes about the same error (± 1.5) also on states that are 2 steps from goal. It turns out, however, that making relatively large mistakes on states close to goal hurts the performance a lot.

A possible explanation seems to be the fact that there is a very large number of states on the edge of the explored space i.e. far from the initial state. These states have similarly low f-value (sum of distance from start and the heuristic estimate). If the heuristic overestimates one of those states even slightly, the state will move far back in the priority queue which significantly delays its expansion. When this happens to a state that really *is* close to goal, the search may get stuck for a long time. This is consistent with results of Helmert and Röger ([45]) who showed that adding even a small additive constant to a perfect heuristic and using that estimate as a heuristic function already makes the search-time exponential.

Plan length	Fast Forward	FF heuristic as feature: FALSE						FF heuristic as feature: TRUE					
		NN heuristic (MSE)			NN heuristic (LogMSE)			NN heuristic (MSE)			NN heuristic (LogMSE)		
problem	heuristic	2	3	4	2	3	4	2	3	4	2	3	4
probBLOCKS-4-0	10	6	6	6	6	6	6	6	6	6	6	6	6
probBLOCKS-5-0	22	14	12	12	12	12	12	12	12	12	12	12	12
probBLOCKS-6-0	24	12	12	12	12	12	12	12	12	12	14	12	12
probBLOCKS-7-0	26	20	20	22	20	20	20	20	22	20	20	20	20
probBLOCKS-8-0	36	18	18	18	18	18	18	22	20	18	18	18	18
probBLOCKS-9-0	48			30		30	30	30	30	30	30	30	30
probBLOCKS-10-0	48			34			34	36	34	34	34	34	34
probBLOCKS-11-0				32			32	32	32	32	32	32	32
probBLOCKS-12-0	42			34			34	34	34	34	34	34	34
probBLOCKS-13-0				42			42	42	42	44	42	42	42
probBLOCKS-14-0				38			38	38	38	38	38	38	38
probBLOCKS-15-0				44			40	42	44	42	42	40	40
probBLOCKS-16-0				54			54		54	54		54	54
probBLOCKS-17-0				48			48	46	48	48	48	48	48
probBLOCKS-18-0							58	60			58	58	58
probBLOCKS-19-0				62			62	62	62	62	62	62	62
probBLOCKS-20-0				60			60	60	60	60	60	60	60
probBLOCKS-21-0				78	78		78		80		78	78	78
probBLOCKS-22-0							72	74	72		72	72	72
probBLOCKS-23-0							76	76	78	78	76	76	76
probBLOCKS-24-0						78	78	78	78		78	78	78
probBLOCKS-25-0				82		82	82		82		82	82	82
probBLOCKS-27-0			86	86	86	86	86	86	86	86	86	86	86
probBLOCKS-28-0				94	92	92	92	94	94	94	92	92	92
probBLOCKS-29-0			90	92	92	92	92		92		92	92	92
probBLOCKS-30-0				96	96	96	96	96	96	96	96	96	96

Figure 5.23: Results of experiments - length of plan found by each heuristic on each problem on *blocks*. Only problems solved by at least one method are included.

In order to improve the search efficiency, the accuracy of the heuristic on states close to goal (i.e. with target of $0 \sim 10$) needs to be much higher. For this reason we've come up with the *LogMSE* loss function.

LogMSE is defined as

$$LogMSE = \frac{\sum \left[\log(y_i + 1) - \log(\hat{y}_i + 1)\right]^2}{n}$$

for $y_i, \hat{y}_i \geq 0$.

As $\left[\log(y_i + 1) - \log(\hat{y}_i + 1)\right]^2 = \log^2\left(\frac{y_i+1}{\hat{y}_i+1}\right)$, *LogMSE* aims to minimize the average of *relative error*, i.e., *the ratio between target and output*. This enforces low absolute error on samples close to goal and tolerates larger absolute error on samples further from goal. Figures 5.32, 5.33 and 5.34 illustrate this behavior.

As figure 5.33 shows, *MSE* (in blue) makes large relative error on states close to goal (i.e. with target of $0 \sim 10$) while *LogMSE* (orange) performs much better on these states but has slightly larger error on states further from goal. Figure 5.34 makes this even more apparent as it provides ratio between targets and average of corresponding outputs.

We believe that achieving low relative error is important in HL applications and *logMSE* or similar loss function should be preferred over *MSE*.

Expanded nodes	Fast Forward	FF heuristic as feature: FALSE						FF heuristic as feature: TRUE					
		NN heuristic (MSE)			NN heuristic (LogMSE)			NN heuristic (MSE)			NN heuristic (LogMSE)		
problem	heuristic	2	3	4	2	3	4	2	3	4	2	3	4
pfile1	2	3	21	2	3	3	2	3	2	2	3	2	2
pfile2	18	146	110	82	149	30	15	19	40	52	11	13	17
pfile3	25	19460	8632	389	15088	86	87	53	73	26	24	31	69
pfile4	63	37250	14097	1103	18744	102	69	629	187	43	68	63	26
pfile5	39		4695	209	183392	78	181	550	1318	97	80	103	208
pfile6	39		28499	1095	181854	196	336	1687	316	1233	350	144	130
pfile7	153		51159	6543		735	442	5630	9129	923	1039	480	127
pfile8	96					650	258		7334	151	806	82	76
pfile9	5698						3946			12668		7615	3222
pfile10	3977					6126	5387					12366	4094
pfile11	315			3181		10089	2835			4048	724	17112	66
pfile12	1469					5957	12546						2030
pfile13	6116												9730
pfile14													160
pfile15													6257
pfile16													1971

Figure 5.24: Results of experiments - number of expanded nodes by each heuristic on each problem on *zenotravel*. Only problems solved by at least one method are included.

Runtime (sec.)	Fast Forward	FF heuristic as feature: FALSE						FF heuristic as feature: TRUE					
		NN heuristic (MSE)			NN heuristic (LogMSE)			NN heuristic (MSE)			NN heuristic (LogMSE)		
problem	heuristic	2	3	4	2	3	4	2	3	4	2	3	4
pfile1	0	0	0	0	0	0	0	0	0	0	0	0	0
pfile2	0	3	2	1	1	0	0	0	0	0	0	0	0
pfile3	0	325	139	22	178	5	4	4	6	3	2	1	3
pfile4	0	527	229	57	203	8	2	20	9	2	4	2	1
pfile5	1		109	7	712	7	5	28	52	8	5	4	6
pfile6	1		750	93	1196	16	11	107	29	76	29	6	7
pfile7	1		518	304		46	10	265	353	54	51	12	4
pfile8	2					75	12		770	20	88	9	7
pfile9	12						224			781		480	201
pfile10	13					856	389					843	376
pfile11	3			762		922	210			722	189	1010	8
pfile12	7					1418	1045						235
pfile13	40												1178
pfile14													86
pfile15													1431
pfile16													1074

Figure 5.25: Results of experiments - runtime for each heuristic on each problem on *zenotravel*. Only problems solved by at least one method are included.

Plan length	Fast Forward	FF heuristic as feature: FALSE						FF heuristic as feature: TRUE					
		NN heuristic (MSE)			NN heuristic (LogMSE)			NN heuristic (MSE)			NN heuristic (LogMSE)		
problem	heuristic	2	3	4	2	3	4	2	3	4	2	3	4
pfile1	1	1	2	1	1	2	1	1	1	1	1	1	1
pfile2	6	6	6	6	6	6	6	6	6	6	6	6	6
pfile3	10	6	8	6	6	7	6	6	8	6	6	6	6
pfile4	11	8	9	8	8	9	8	11	11	8	8	8	8
pfile5	12		11	11	11	11	11	11	11	11	11	11	11
pfile6	15		12	11	12	11	11	12	12	11	11	11	11
pfile7	16		15	15		15	15	16	15	15	16	15	15
pfile8	16					11	11		11	11	11	11	11
pfile9	27						21			21		21	21
pfile10	28					22	22					22	22
pfile11	18			14		14	14			14	14	14	14
pfile12	28					23	21						21
pfile13	39												29
pfile14													30
pfile15													43
pfile16													44

Figure 5.26: Results of experiments - length of plan found by each heuristic on each problem on *zenotravel*. Only problems solved by at least one method are included.

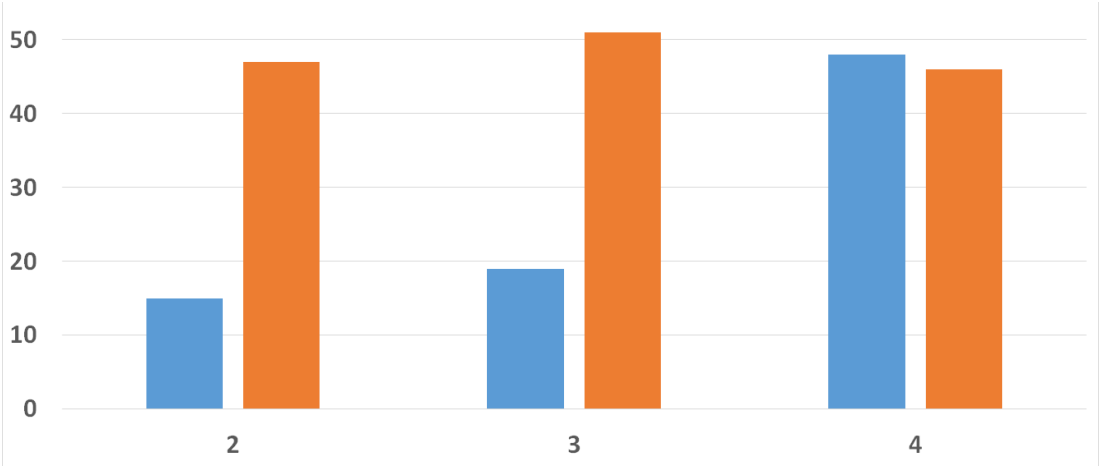

Figure 5.27: Number of problems solved in *blocks* domain (sum of both *MSE* and *LogMSE*). On the X-axis there is α value, blue columns correspond to networks trained without using h_{FF} as feature, orange columns show networks trained with h_{FF} included.

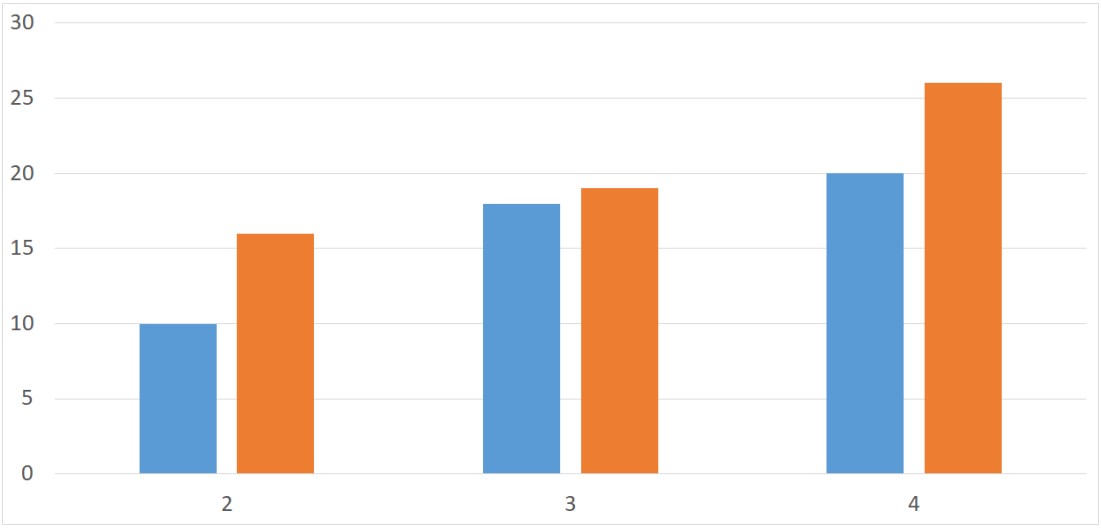

Figure 5.28: Number of problems solved in *zenotravel* domain (sum of both *MSE* and *LogMSE*). On the X-axis there is α value, blue columns correspond to networks trained without using h_{FF} as feature, orange columns show networks trained with h_{FF} included.

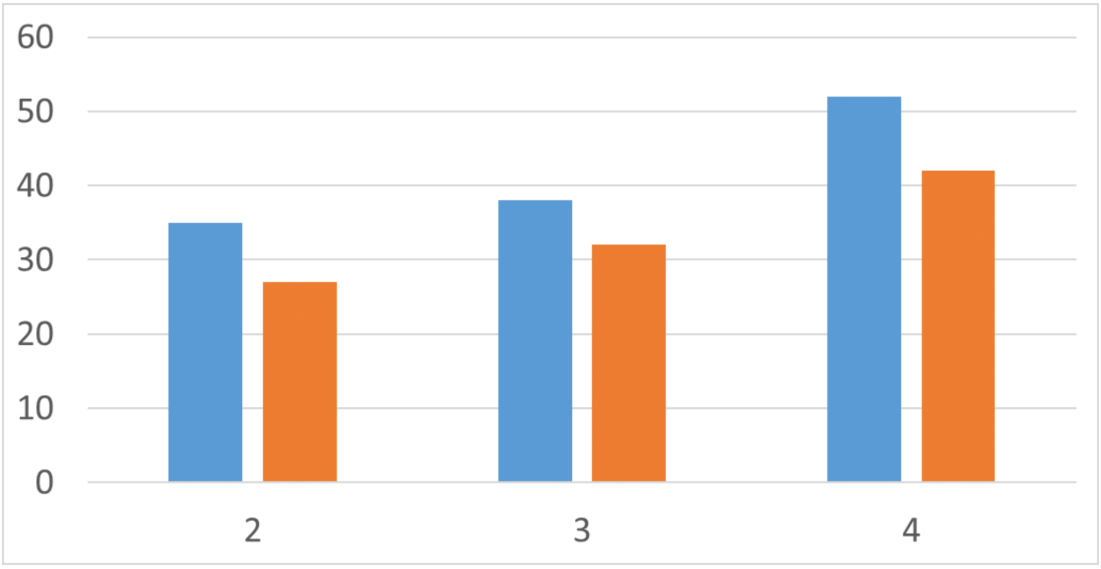

Figure 5.29: Number of problems solved by heuristics trained by *LogMSE* (blue columns) and *MSE* (orange columns) for various values of α in *blocks*.

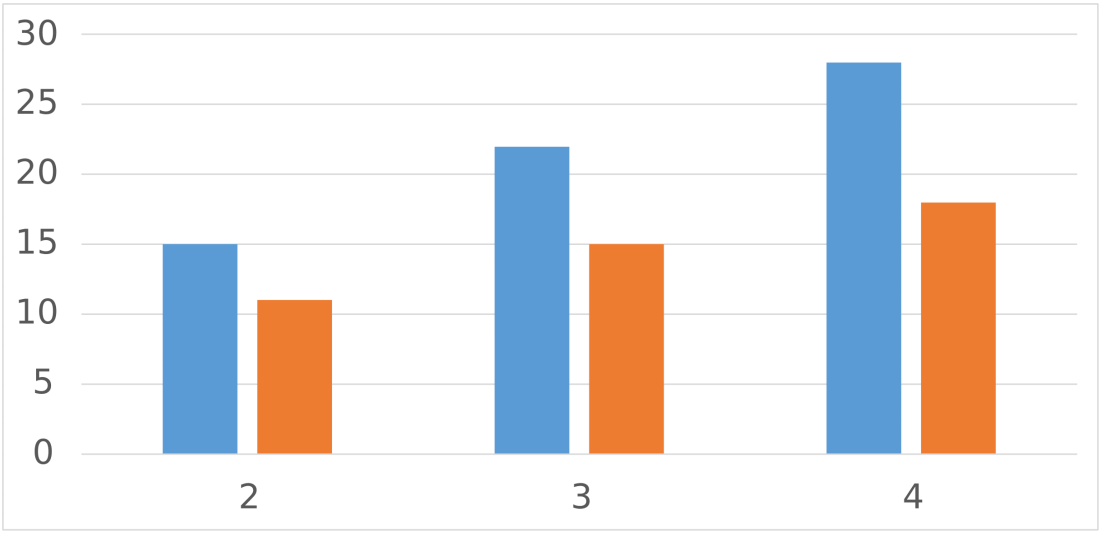

Figure 5.30: Number of problems solved by heuristics trained by *LogMSE* (blue columns) and *MSE* (orange columns) for various values of α in *zenotravel*.

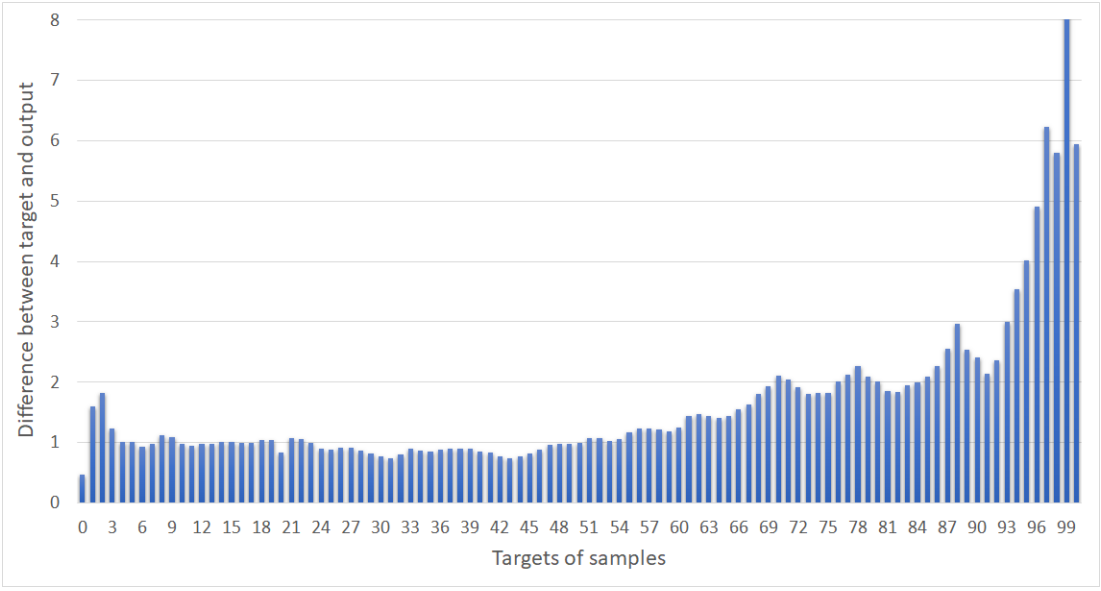

Figure 5.31: Absolute error on samples with different targets. On the X-axis there is target - i.e. real goal distance, height of the column shows average of absolute error ($\left|Y_i - \hat{Y}_i\right|$) over all samples that fall into that category. Results of training with *MSE*.

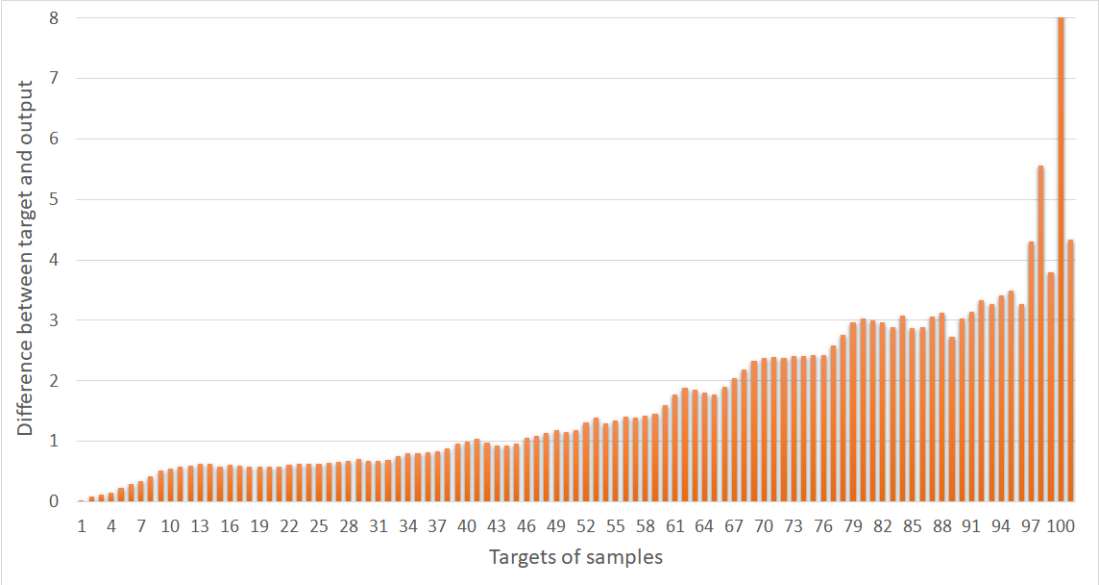

Figure 5.32: Absolute error on samples with different targets. On the X-axis there is target - i.e. real goal distance, height of the column shows average of absolute error ($\left|Y_i - \hat{Y}_i\right|$) over all samples that fall into that category. Results of training with *LogMSE*.

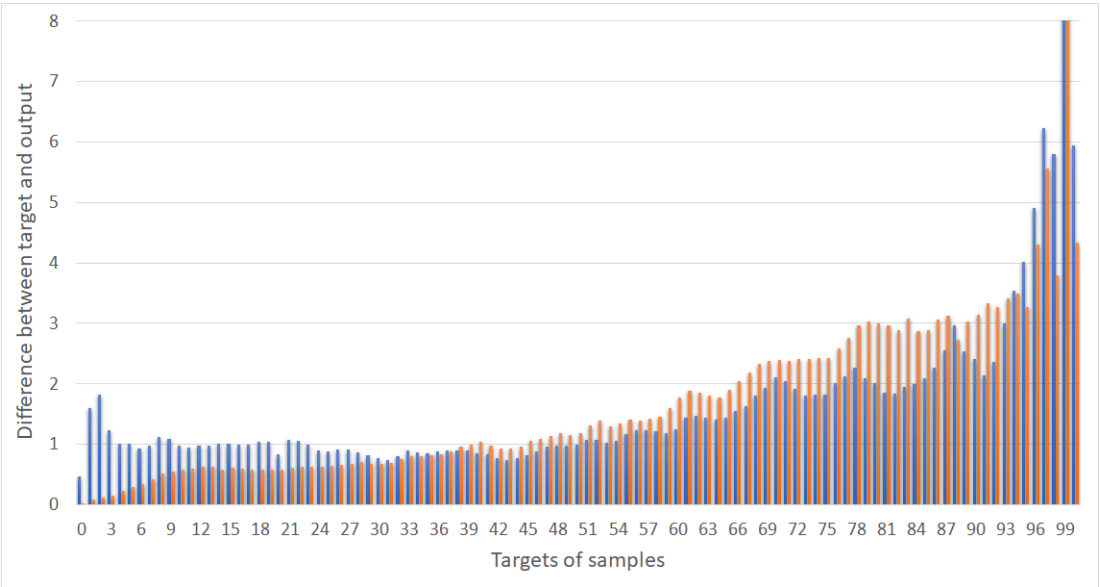

Figure 5.33: Figures 5.31 and 5.32 combined.

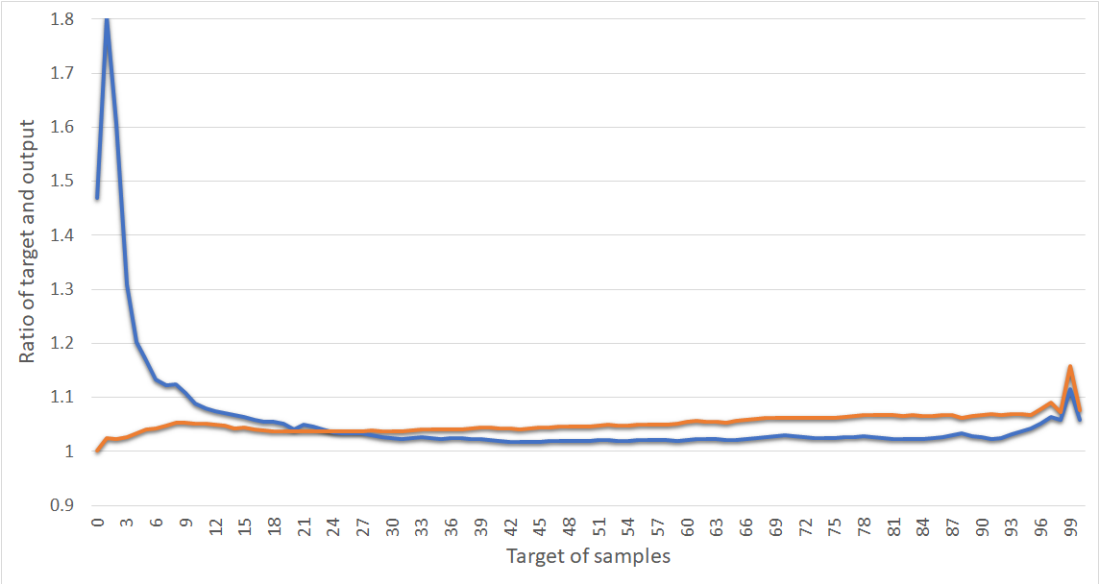

Figure 5.34: Relative error on samples with different targets. On the X-axis there is target - i.e. distance to go, on the y-axis there is average of relative error ($\max(\frac{y+1}{\hat{y}+1}, \frac{\hat{y}+1}{y+1})$) of corresponding samples. Blue line represents networks trained by *MSE*, orange line the ones trained by *LogMSE*.

6. Performance Guarantees

In this chapter, we provide theoretical performance guarantees for heuristics constructed by HL approaches. We define the notion of Stochastic heuristic and we state and prove it's theoretical properties.

We then show that under some reasonable assumptions, learned heuristics can be modeled via the stochastic heuristic framework. The presented performance bounds are applicable to any learned heuristic, not just to those created by our method.

6.1 Motivation

Learned heuristics work well in practice as proved by our experiments as well as by other authors ([80, 52]) but in general they are not admissible nor ϵ-admissible for any reasonable ϵ. They don't provide any theoretical guarantees on the performance like the hand-coded heuristics do via theorems 1 and 2. These theorems are not applicable to learned heuristics as they require the heuristic error to be bounded on *all* states.

ML techniques don't provide these types of guarantees. After the ML model is trained, typically we get a picture similar to one in figure 5.16. The model achieves low error on *average* but there could be a very small percentage of inputs on which the error is high. Standard heuristics are designed to provide limit on *maximum* of error but they tell us nothing about *average* error. Learned heuristics, on the other hand, provide low *average* error but there is no bound on the *maximum*.

We would like to leverage the fact that on *average* the error is low in order to generalize theorem 2 in a probabilistic manner, i.e., guaranteeing that the quality of the solution will be good with some high probability.

Unfortunately, requirements in theorem 2 are tight in a sense that even if a single state would violate the requirements, there would exist a problem for which the stated bound does not hold. Formally:

$$\forall \epsilon, h \; \exists P, s' \in S^P : h(s') > \epsilon \cdot h^*(s') \Rightarrow A_h(s_0^P) > \epsilon \cdot h^*(s_0^P)$$

, where $A_h(s_0^P)$ is cost of solution that A^* algorithm finds from s_0 and $h^*(s_0)$ is the cost of optimal solution. (See definitions 10 and 12.)

This is problematic for our probabilistic generalization as we have no control over distribution of problems, i.e., we cannot argue that the counterexample mentioned above is "very unlikely to occur". For this reason, arguing that the heuristic works well on "most" problems is not enough. We must assume that the problem will be given by an adversary and provide a worst-case probabilistic bound that will hold for *every* problem.

6.2 Stochastic Heuristics

We will now define a formal notion of *Stochastic heuristic* and formulate some of its properties. We will later show how heuristics obtained by HL can be modeled as stochastic heuristics.

Definition 27 (Stochastic heuristic)**.** *Given a problem P, a **stochastic heuristic** H is a set of independent non-negative random variables $H(s_i)$ for each state $s_i \in S^P$.*

When H is used in search, we first sample each $H(s_i)$ to obtain a *fixed* estimate for each state, i.e. a standard heuristic. Then we use the standard heuristic for search so during the search, there is no randomness involved. If the algorithm encounters the same state several times, is always assigns it the same estimate. Running another search on the same problem could of course produce different results as we could obtain a different standard heuristic via the initial sampling. When analyzing properties of a stochastic heuristic, we always address the situation *before* the initial sampling hence both $H(s)$ and $A_H(s)$ are random variables (RVs).

6.2.1 Properties of stochastic heuristics

We formulate and prove three theorems that provide bounds on solution quality when using A^* algorithm with stochastic heuristic. These theorems are probabilistic generalizations of theorem 2. In general, we show how distributions of $H(s_i)$ affect distribution of $A_H(s_0)$.

We work with a restricted planning problem, that is a problem that contains exactly one goal state and exactly one optimal path from s_0 to the goal, cost of each action is 1 and the initial state is not a goal state.

Theorem 4. *Let P be a planning problem and H be a stochastic heuristic. If $\forall s \in S^P : \mathsf{E}[H(s)] = h^*(s)$, then $\forall c > 1 : \mathsf{P}[A_H(s_0) \geq c \cdot (h^*(s_0))^2] < \frac{1}{c}$*

Proof. Let's first state several inequalities that we will use in the proof.

Lemma 5 (Markov's inequality)**.** *Let X be a random variable, such that $X > 0$, then $\forall a > 0 : \mathsf{P}[X \geq a] \leq \frac{\mathsf{E}[X]}{a}$.*

Proof. See [32]. □

Lemma 6 (Boole's inequality)**.** *Let A_i be events, then $\mathsf{P}[\bigcup A_i] \leq \sum \mathsf{P}[A_i]$*

Proof. See Theorem 1.2.11 in [10]. □

Lets denote by $Opt(s_0)$ the optimal path from s_0 to the goal (a sequence of states) except the initial and the goal state. Since we work with unit-cost actions, $|Opt(s_0)| = h^*(s_0) - 1$ and $\forall s_i \in Opt(s_0) : h^*(s_i) = h^*(s_0) - i$.
By a contraposition of theorem 2, we have

$$\forall \epsilon \geq 1 : A_H(s_0) > \epsilon \cdot h^*(s_0) \Rightarrow \tag{6.1}$$
$$\exists s_i \in Opt(s_0) : H(s_i) > \epsilon \cdot h^*(s_i) \tag{6.2}$$

hence

$$\mathsf{P}\left[A_H(s_0) > \epsilon \cdot h^*(s_0)\right] \leq \tag{6.3}$$

$$\mathsf{P}\left[\exists s_i \in Opt(s_0) : H(s_i) > \epsilon \cdot h^*(s_i)\right] \leq \tag{6.4}$$

$$\sum_{s_i \in Opt(s_0)} \mathsf{P}\left[H(s_i) > \epsilon \cdot h^*(s_i)\right] \tag{6.5}$$

(3) $\leq$ (4) comes from (1) $\Rightarrow$ (2), while (4) $\leq$ (5) can be achieved by applying Boole's inequality.

Now, Markov's inequality states that

$$\forall s_i : \mathsf{P}[H(s_i) \geq \epsilon \cdot h^*(s_i)] \leq \frac{\mathsf{E}\left[H(s_i)\right]}{\epsilon \cdot h^*(s_i)} = \frac{h^*(s_i)}{\epsilon \cdot h^*(s_i)} = \frac{1}{\epsilon} \tag{6.6}$$

By substituting (6) to (5) we get:

$$\sum_{s_i \in Opt(s_0)} \mathsf{P}[H(s_i) > \epsilon \cdot h^*(s_i)] \leq$$

$$\sum_{s_i \in Opt(s_0)} \frac{1}{\epsilon} = |Opt(s_0)| \cdot \frac{1}{\epsilon} \leq \frac{h^*(s_0)}{\epsilon}$$

We've created a chain of inequalities. When we put together the first and the last element of the chain, we get

$$\forall \epsilon > 1 : \mathsf{P}\left[A_H(s_0) > \epsilon \cdot h^*(s_0)\right] < \frac{h^*(s_0)}{\epsilon}$$

Now for given $c > 1$, we set $\epsilon = c \cdot h^*(s_0)$ which gives us the required bound. □

Bound in theorem 4 is quite loose as it uses $[h^*(s_0)]^2$. If we take variance of $H(s)$ into account, we can come up with a tighter bound.

Theorem 7. *Let P be a search problem and H be a stochastic heuristic such that $\forall s \in S^P : \mathsf{E}[H(s)] = h^*(s)$, $var[H(s)] = \sigma^2 < \infty$. Then*

$$\forall c > 1 : \mathsf{P}[A_H(s_0) > c \cdot (h^*(s_0))] < \frac{\sigma}{c-1} \cdot \frac{\pi}{2}$$

Proof. We will use the following inequalities in the proof.

Lemma 8 (Cantelli's inequality)**.** *Let X be a random variable, such that $\mathsf{E}[X] = \mu < \infty$ and $var[X] = \sigma^2 < \infty$, then $\forall a > 0 : \mathsf{P}[X - \mu \geq a] \leq \frac{\sigma^2}{\sigma^2 + a^2}$*

Proof. See corollary of Theorem 1 in [32]. □

Lemma 9 (Integral approximation of a sum)**.** *Let f be a non-decreasing function on $[a-1, b+1]$, then:*

$$\int_{a-1}^{b} f(s)\ ds \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(s)\ ds$$

Proof. See *Remark 12.105* in [60]. □

We proceed in the same way as in the previous proof up to equation (5). We have

$$\forall \epsilon > 1 : \mathsf{P}\left[A_H(s_0) > \epsilon \cdot h^*(s_0)\right] < \sum_{s_i \in Opt(s_0)} \mathsf{P}\left[H(s_i) > \epsilon \cdot h^*(s_i)\right]$$

Lets denote $z = h^*(s_0)$. Then $|Opt(s_0)| = z - 1$ and $\forall s_i \in Opt(s_0) : h^*(s_i) = z - i$.

When we use Cantelli's inequality, we have:

$$\mathsf{P}[H(s_i) \geq \epsilon \cdot h^*(s_i)] = \mathsf{P}[H(s_i) \geq \epsilon \cdot (z - i)] \leq \frac{\sigma^2}{\sigma^2 + (\epsilon - 1)^2 (z - i)^2}$$

When we substitute that to the sum, we get

$$\sum_{s_i \in Opt(s_0)} \mathsf{P}\left[H(s_i) > \epsilon \cdot h^*(s_i)\right] < \sum_{i=1}^{z-1} \frac{\sigma^2}{\sigma^2 + (\epsilon - 1)^2 (z - i)^2} \leq$$

Using Lemma 9, we get

$$\leq \int_1^z \frac{\sigma^2}{\sigma^2 + (\epsilon - 1)^2 (z - i)^2} \, di =$$

$$\frac{\sigma}{\epsilon - 1} \arctan \frac{(z - 1)(\epsilon - 1)}{\sigma} \leq \frac{\sigma}{\epsilon - 1} \cdot \frac{\pi}{2}$$

Now given $c > 1$, we just set $\epsilon = c$.

□

Theorem 7 is a direct generalization of theorem 2 as it imposes the same bound on $A_h(s)$. The probability bound, however, is quite loose hence the theorem is only applicable for large c. Specifically for $c > \frac{\sigma \cdot \pi}{2} + 1$. Since σ is the standard deviation of $H(s)$, is could be quite large in practice: between 5 and 10 or even larger depending on the problem.

Theorems 4 and 7 make no assumptions about distribution of $H(s_i)$, i.e. they work for any distribution. They don't even require that $H(s_i)$ are identically distributed. A much better bound can be obtained for specific distributions.

Theorem 10. *Let P be a search problem and H be a stochastic heuristic such that $\forall s \in S^P : H(s) \sim N(h^*(s), \sigma^2), \sigma^2 < \infty$. Then*

$$\forall c > 1 : \mathsf{P}[A_H(s_0) \geq c \cdot (h^*(s_0))] < \exp\left\{-\frac{(c-1)^2}{2\sigma^4}\right\} \frac{\sigma^2}{\sqrt{2\pi}(c-1)}$$

Proof. We again make use of several well known notions and inequalities.

Definition 28 (Gauss error function)**.** *The **Gauss error function** erf is defined as*

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty \exp\left\{-t^2\right\} dt$$

By erfc we denote the ***complementary error function****:*

$$erfc(x) = 1 - erf(x)$$

Erfc can be used to calculate cumulative distribution function for normally distributed random variables.

Lemma 11 (CDF of normally distributed random variable)**.** *Let* $X \sim N(\mu, \sigma^2)$*, then* $\forall q : \mathsf{P}[X > q] = 1 - \frac{1}{2} erfc\left(\frac{\mu - q}{\sqrt{2\sigma^2}}\right)$

There is no closed-form formula for $erf(x)$, so various approximations are used:

Lemma 12 (Bounds on Gauss error function)**.** *For all* $x > 0$*:*

$$\frac{2\exp\{-x^2\}}{\sqrt{\pi}\left(\sqrt{x^2+2}+x\right)} < erfc(x) < \frac{\exp\{-x^2\}}{\sqrt{\pi}x}$$

Proof. See [12], sections *III* and *IV*. □

We again proceed in the same way as in the proof of theorem 4 up to equation (5) which gives us

$$\forall \epsilon > 1 : \mathsf{P}\left[A_H(s_0) > \epsilon \cdot h^*(s_0)\right] < \sum_{s_i \in Opt(s_0)} \mathsf{P}\left[H(s_i) > \epsilon \cdot h^*(s_i)\right] =$$

Using $z = h^*(s_0)$, $|Opt(s_0)| = z - 1$ and $\forall s_i \in Opt(s_0) : h^*(s_i) = z - i$, we get

$$= \sum_{i=1}^{z-1} \mathsf{P}\left[H(s_i) > \epsilon \cdot (z - i)\right] \leq$$

By applying Lemma 9 we get

$$\leq \int_1^z \mathsf{P}\left[H(s_i) > \epsilon \cdot (z - i)\right] di \leq$$

we can just calculate this directly using Lemmata 11 and 12 to obtain

$$\leq \frac{\sigma^2}{\exp\left\{\frac{(\epsilon-1)^2}{2\sigma^4}\right\}\sqrt{2\pi}(\epsilon - 1)} - \frac{\exp\left\{-\frac{(\epsilon-1)^2}{2\sigma^4}\right\}}{\sqrt{\pi}\left(\frac{\epsilon-1}{\sqrt{2}\sigma^2} + \sqrt{\frac{(\epsilon-1)^2}{2\sigma^4}+2}\right)}$$

Now we can drop the second term which is negative hence by doing so we only increase the value. For given $c > 1$, we set $\epsilon = c$. That gives us the required bound. □

Theorem 10 is based on a strong assumption of normality but it also provides a very strong bound: the probability of producing a solution whose cost is c-times greater than the optimal cost is exponentially small with respect to c^2.

6.2.2 Learned heuristics as *Stochastic heuristics*

In the rest of this chapter, we will describe how learned heuristics can be modeled as *stochastic heuristics* and we will discuss whether assumptions of theorems 4, 7 and 10 actually hold in HL applications.

Stochastic heuristic assigns a random variable to each state. The learned heuristic, on the other hand, is represented by a neural network. Let θ be a vector of trainable parameters of the network, i.e. weights and biases. During training, θ is adjusted to minimize the given loss function. The training algorithm is randomized, so the resulting parameter vector θ is a random variable.

We can model the function represented by the NN as a stochastic heuristic in the following way. Lets denote by $H(s) = NN_\theta(s)$ the heuristic value for a state s, i.e. the output of a neural network with parameters θ on a state s. Since θ is random variable, $H(s)$ is a random variable as well. Once the NN is trained, however, it operates in a deterministic manner.

The stochastic heuristic model applies to the neural net *before* it is trained. Training the network (and fixing θ) corresponds to sampling the stochastic heuristic and obtaining a standard heuristic. This modeling approach is necessary because of the adversary-argument mentioned before.

After the NN is trained, it is possible to analyze it, discover its weaknesses and come up with an adversary example: a planning problem on which the learned heuristic performs poorly. We can overcome this by defining the stochastic heuristic in the way we did, i.e. analyzing behavior of the net *before* it is trained. First the adversary produces the problem and only after that the stochastic heuristic is sampled. Since the adversary has no control over the sampling, the probabilistic bounds will hold.

When stating the probabilistic bounds, we don't average over problems but rather over all possible outcomes of the randomized training algorithm. When we are given an adversary example, we can just re-train the network using the same training data and chances are that the new network will no longer have any difficulties with that example.

6.2.3 Practical applicability

Our theorems are based on several assumptions: independence of $H(s_i)$ and $H(s_j)$ for $s_i \neq s_j$, unbiasedness, i.e. $\mathsf{E}[H(s)] = h^*(s)$ and non-negativity:
$\forall s : \mathsf{P}[H(s) < 0] = 0$.

We can easily guarantee non-negativity of estimates by restricting outputs of the NN only to positive values, i.e. replacing any negative output by zero. Assumptions of independence and unbiasedness of $H(s_i)$ are more tricky, but can be justified by analyzing the bias-variance tradeoff for NNs ([39]). NNs are universal approximators ([50]) and in general they have high *variance* and low *bias* hence for a large enough network, $H(s)$ should be unbiased and $\forall s_i, s_j : H(s_i)$ and $H(s_j)$ should be close to independent.

Normality

The normality assumption can't be guaranteed in practice. We used it mostly to demonstrate that a very tight bound can be proved when we know the distribution

of outputs with respect to the randomized training algorithm. In general, such distribution is very difficult to analyze. The distribution of $h^*(s_i)$ with respect to *features of* s_i is domain specific hence distribution of $H(s_i)$ will be domain specific as well.

We don't require that all $H(s_i)$ are identically distributed. In theorems 7 and 10 we're assuming that $\text{var}[H(s_i)]$ is the same for all states but even if it wasn't, the theorem would still hold with an identical proof. In fact, we can just use $\sigma^2 = \max_{s_i \in S} \text{var}[H(s_i)]$. If the actual variance is lower for some states, it will only improve the bound.

Theorems stated here are not tied to our particular HL framework. They can be applied to any HL system and also to other kinds of heuristics that were constructed by stochastic optimization algorithms like Monte-Carlo Tree Search, heuristic-selection approaches, genetic algorithms, hyper-heuristics and others, as long as they fulfill the requirements of independence, unbiasedness and non-negativity.

Conclusion

In this thesis, we analyzed the issue of utilizing ML techniques to improve efficiency of heuristic forward-search algorithms. We followed two lines: using data analysis to adjust existing hand-coded heuristics and creating new heuristics from scratch which is the main topic of the thesis.

We have developed a way to automatically adjust any given heuristic based on a set of training data. The adjustments are domain-specific and can leverage domain knowledge as well as users' preferences regarding search-time vs. solution quality trade-off. The process is fully automatic and much more flexible than simple weighting.

We presented a technique to automatically construct a strong heuristic for a given planning domain. Our technique is domain-independent and can extract knowledge about any STRIPS domain using a given set of training problems. The knowledge in represented by a trained neural network. Our technique outperforms a popular domain-independent heuristic h_{FF} in both number of problems solved and solution quality on two challenging planning domains.

We analyzed the problem from the ML perspective, described how choice of training data affects generalization, and how an inappropriate choice of loss-function causes poor search-performance of an otherwise well-trained model. We showed that the Mean Squared Error – the most common loss function – is not appropriate for heuristic learning tasks and we presented a better alternative.

We pointed out strong correspondences between settings of the training algorithm and properties of the learned heuristic. We also proposed a categorization of HL approaches based on different generalization capabilities of the heuristic.

We've developed a novel technique for extracting features from states of planning problems where we encode the state directly without using existing hand-coded heuristics. The method allows to compute features incrementally during the forward-search which is very useful in planning application. An informed and incrementally computable features extractor have been one of the few missing pieces to making domain-independent heuristic learning competitive with the state-of-the-art.

Last but not least, we have provided theoretical bounds on solution quality for learned heuristics. These can be considered probabilistic generalizations of the well-known bounds for weighted A*.

Our approach falls into category of *zero-learning* as it works without any human-knowledge initially added. In general, hand-coded tools provide *worst-case* performance guarantees and are *explainable* while ML approaches provide better *average* performance. The HL framework that we've developed combines advantages of both as it works well in practice and we provided theoretical performance guarantees that make the system trustworthy.

We believe that machine learning still has a great undiscovered potential to improve heuristic design as well as other aspects of planning and combinatorial search in general and that we will be seeing more of these attempts in the future.

Bibliography

[1] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Bootstrap learning of heuristic functions. In Ariel Felner and Nathan R. Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010*. AAAI Press, 2010. URL `http://aaai.org/ocs/index.php/SOCS/SOCS10/paper/view/2071`.

[2] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16), 2011. ISSN 0004-3702.

[3] Edward A. Bender, E. Rodney Canfield, and Brendan D. McKay. The asymptotic number of labeled connected graphs with a given number of vertices and edges. *Random Structures & Algorithms*, 1(2):127–169, 1990. doi: 10.1002/rsa.3240010202. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/rsa.3240010202`.

[4] Rafael Vales Bettker, Pedro Minini, A. G. Pereira, and M. Ritt. Understanding sample generation strategies for learning heuristic functions in classical planning. *CoRR*, abs/2211.13316, 2022. doi: 10.48550/arXiv.2211.13316. URL `https://doi.org/10.48550/arXiv.2211.13316`.

[5] Francis Bisson, Hugo Larochelle, and Froduald Kabanza. Using a recursive neural network to learn an agent's decision model for plan recognition. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[6] Blai Bonet and Malte Helmert. Strengthening landmark heuristics via hitting sets. *Frontiers in Artificial Intelligence and Applications*, 215:329–334, 06 2010. doi: 10.3233/978-1-60750-606-5-329.

[7] Martim Brandao, Amanda Jane Coles, Andrew Coles, and Jörg Hoffmann. Merge and shrink abstractions for temporal planning. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24, 2022*, pages 16–25. AAAI Press, 2022. URL `https://ojs.aaai.org/index.php/ICAPS/article/view/19781`.

[8] Robert Brunetto and Otakar Trunda. Deep heuristic-learning in the rubik's cube domain: an experimental evaluation. In J. Hlaváčová, editor, *Proceedings of the 17th ITAT Conference Information Technologies - Applications and Theory*, pages 57–64. CreateSpace Independent Publishing Platform, 2017. ISBN 978-1974274741.

[9] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[10] G. Casella and R.L. Berger. *Statistical Inference*. Cengage Learning, 2021. ISBN 9780357753132. URL https://books.google.cz/books?id=FAUVEAAAQBAJ.

[11] Isabel Cenamor, Tomás De La Rosa, and Fernando Fernández. Learning predictive models to configure planning portfolios. In *Proceedings of the 4th workshop on Planning and Learning (ICAPS-PAL 2013)*, 2013.

[12] Seok-Ho Chang, Pamela C Cosman, and Laurence B Milstein. Chernoff-type bounds for the gaussian error function. *IEEE Transactions on Communications*, 59(11):2939–2944, 2011.

[13] Hung-Che Chen and Jyh-Da Wei. Using neural networks for evaluation in heuristic search algorithm. In *AAAI*, 2011.

[14] Joseph Culberson. Sokoban is pspace-complete. Technical report, 1997.

[15] Andrea Dittadi, Frederik K. Drachmann, and Thomas Bolander. Planning from pixels in atari with learned symbolic representations. *CoRR*, abs/2012.09126, 2020. URL https://arxiv.org/abs/2012.09126.

[16] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. Red–black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence*, 221:73 – 114, 2015. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2014.12.008. URL http://www.sciencedirect.com/science/article/pii/S0004370214001581.

[17] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[18] Filip Dvořák and Roman Barták. Integrating time and resources into planning. In *22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 2, pages 71–78. IEEE, 2010.

[19] Stefan Edelkamp. Optimal symbolic pddl3 planning with mips-bdd. *5th International Planning Competition Booklet (IPC-2006)*, pages 31–33, 2006.

[20] Stefan Edelkamp, Stefan Schroedl, and Sven Koenig. *Heuristic Search: Theory and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010. ISBN 0123725127.

[21] Stefan Edelkamp, Peter Kissmann, and Álvaro Torralba. BDDs strike back (in AI planning). In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 4320–4321. AAAI Press, 2015. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9834.

[22] Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith, editors, *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, pages 88–97. AAAI Press,

2017. URL https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15734.

[23] Patrick Ferber, Malte Helmert, and Jörg Hoffmann. Neural network heuristics for classical planning: A study of hyperparameter space. In *ECAI 2020 - 24th European Conference on Artificial Intelligence*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2346–2353. IOS Press, 2020. doi: 10.3233/FAIA200364. URL https://doi.org/10.3233/FAIA200364.

[24] Patrick Ferber, Florian Geißer, Felipe W. Trevizan, Malte Helmert, and Jörg Hoffmann. Neural network heuristic functions for classical planning: Bootstrapping and comparison to other methods. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24, 2022*, pages 583–587. AAAI Press, 2022. URL https://ojs.aaai.org/index.php/ICAPS/article/view/19845.

[25] Maximilian Fickert and Jörg Hoffmann. Online relaxation refinement for satisficing planning: On partial delete relaxation, complete hill-climbing, and novelty pruning. *Journal of Artificial Intelligence Research*, 73:67–115, 2022.

[26] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3): 189–208, 1971. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(71)90010-5. URL https://www.sciencedirect.com/science/article/pii/0004370271900105.

[27] Michael Fink. Online learning of search heuristics. In Marina Meila and Xiaotong Shen, editors, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics, AISTATS 2007, San Juan, Puerto Rico, March 21-24, 2007*, volume 2 of *JMLR Proceedings*, pages 114–122. JMLR.org, 2007. URL http://proceedings.mlr.press/v2/fink07a.html.

[28] Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5554–5561. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/771. URL https://doi.org/10.24963/ijcai.2019/771.

[29] Sankalp Garg, Aniket Bajpai, and Mausam. Size independent neural transfer for RDDL planning. In J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava, editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, pages 631–636. AAAI Press, 2019. URL https://ojs.aaai.org/index.php/ICAPS/article/view/3530.

[30] Clement Gehring, Masataro Asai, Rohan Chitnis, Tom Silver, Leslie Kaelbling, Shirin Sohrabi, and Michael Katz. Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 588–596, 2022.

[31] Cedric Geissmann. Learning heuristic functions in classical planning. Master's thesis, University of Basel, Switzerland, 2015.

[32] B. K. Ghosh. Probability inequalities related to markov's theorem. *The American Statistician*, 56(3):186–190, 2002. ISSN 00031305. URL `http://www.jstor.org/stable/3087296`.

[33] Daniel Gilon, Ariel Felner, and Roni Stern. Dynamic potential search—a new bounded suboptimal search. In *Ninth Annual Symposium on Combinatorial Search*, 2016.

[34] Pawel Gomoluch, Dalal Alrajeh, Alessandra Russo, and Antonio Bucchiarone. Towards learning domain-independent planning heuristics. *arXiv preprint arXiv:1707.06895*, 2017.

[35] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[36] Edward Groshev, Maxwell Goldstein, et al. Learning generalized reactive policies using deep neural networks. *Symposium on Integrating Representation, Reasoning, Learning, and Execution for Goal Directed Autonomy*, 2017.

[37] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. An investigation of model-free planning. In *International Conference on Machine Learning*, pages 2464–2473. PMLR, 2019.

[38] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019. doi: 10.2200/S00900ED2V01Y201902AIM042.

[39] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[40] Zhibo He, Chenguang Liu, Xiumin Chu, Rudy R Negenborn, and Qing Wu. Dynamic anti-collision a-star algorithm for multi-ship encounter situations. *Applied Ocean Research*, 118:102995, 2022.

[41] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006. ISSN 1076-9757. doi: 10.1613/jair.1705. URL `https://doi.org/10.1613/jair.1705`.

[42] Malte Helmert. *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-77722-9.

[43] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5):503–535, 2009. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2008.10.013. URL https://www.sciencedirect.com/science/article/pii/S0004370208001926. Advances in Automated Plan Generation.

[44] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'09, page 162–169. AAAI Press, 2009. ISBN 9781577354062.

[45] Malte Helmert and Gabriele Röger. How good is almost perfect? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, page 944–949. AAAI Press, 2008. ISBN 9781577353683.

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

[47] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *J. ACM*, 61(3), June 2014. ISSN 0004-5411. doi: 10.1145/2559951. URL https://doi.org/10.1145/2559951.

[48] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001. doi: 10.1613/jair.855. URL https://doi.org/10.1613/jair.855.

[49] Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. On guiding search in HTN planning with classical planning heuristics. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6171–6175. ijcai.org, 2019. doi: 10.24963/ijcai.2019/857. URL https://doi.org/10.24963/ijcai.2019/857.

[50] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90009-T. URL http://www.sciencedirect.com/science/article/pii/089360809190009T.

[51] Sergio Jiménez, Tomás De la Rosa, Susana Fernández, Fernando Fernández, and Daniel Borrajo. A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(4):433–467, 2012.

[52] Rushang Karia and Siddharth Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 8064–8073. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16983.

[53] Michael Katz. Red-black heuristics for planning tasks with conditional effects. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33 (01):7619–7626, Jul. 2019. doi: 10.1609/aaai.v33i01.33017619. URL https://ojs.aaai.org/index.php/AAAI/article/view/4755.

[54] Michael Katz. Red-black heuristics for planning tasks with conditional effects. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33: 7619–7626, 07 2019. doi: 10.1609/aaai.v33i01.33017619.

[55] Thomas Keller, Florian Pommerening, Jendrik Seipp, Florian Geißer, and Robert Mattmüller. State-dependent cost partitionings for cartesian abstractions in classical planning. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 2016.

[56] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[57] Peter Kissmann and Stefan Edelkamp. Improving cost-optimal domain-independent symbolic planning. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[58] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61, 2018.

[59] Nir Lipovetzky, Miquel Ramirez, and Hector Geffner. Classical planning with simulators: Results on the atari video games. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 1610–1616. AAAI Press, 2015. ISBN 9781577357384.

[60] C. Mariconda and A. Tonolo. *Discrete Calculus: Methods for Counting*. UNITEXT. Springer International Publishing, 2016. ISBN 9783319030388. URL https://books.google.cz/books?id=37iiDQAAQBAJ.

[61] Mario Martín and Hector Geffner. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, 20(1):9–19, 2004.

[62] Felipe Meneguzzi and Ramon Fraga Pereira. A survey on goal recognition as planning. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event /*

Montreal, Canada, 19-27 August 2021, pages 4524–4532. ijcai.org, 2021. doi: 10.24963/ijcai.2021/616. URL https://doi.org/10.24963/ijcai.2021/616.

[63] Zied Mnasri, Stefano Rovetta, and Francesco Masulli. Anomalous sound event detection: A survey of machine learning based methods and applications. *Multimedia Tools and Applications*, 81(4):5537–5586, 2022.

[64] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337), 2017. ISSN 1095-9203. doi: 10.1126/science.aam6960. URL http://dx.doi.org/10.1126/science.aam6960.

[65] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 1558608567.

[66] Jennifer M. Nelson and Rogelio E. Cardona-Rivera. Partial-order, partially-seen observations of fluents or actions for plan recognition as planning. *CoRR*, abs/1911.05876, 2019. URL http://arxiv.org/abs/1911.05876.

[67] Stefan O'Toole, Miquel Ramirez, Nir Lipovetzky, and Adrian R Pearce. Sampling from pre-images to learn heuristic functions for classical planning. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, pages 308–310, 2022.

[68] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley, 1984. ISBN 9780201055948. URL https://books.google.cz/books?id=0XtQAAAAMAAJ.

[69] André Grahl Pereira, Marcus Ritt, and Luciana S. Buriol. Optimal sokoban solving using pattern databases with specific domain knowledge. *Artif. Intell.*, 227:52–70, 2015. doi: 10.1016/j.artint.2015.05.011. URL https://doi.org/10.1016/j.artint.2015.05.011.

[70] André Grahl Pereira, Robert Holte, Jonathan Schaeffer, Luciana S. Buriol, and Marcus Ritt. Improved heuristic and tie-breaking for optimally solving sokoban. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 662–668. IJCAI/AAAI Press, 2016. URL http://www.ijcai.org/Abstract/16/100.

[71] Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet. Lp-based heuristics for cost-optimal planning. In *Proceedings of the Twenty-Fourth International Conferenc on International Conference on Automated Planning and Scheduling*, ICAPS'14, page 226–234. AAAI Press, 2014. ISBN 9781577356608.

[72] Raymond Reiter. *Knowledge in action*. Mit Press, 2001. ISBN 978-0-262-52700-2.

[73] Or Rivlin, Tamir Hazan, and Erez Karpas. Generalized planning with deep reinforcement learning. *CoRR*, abs/2005.02305, 2020. URL `https://arxiv.org/abs/2005.02305`.

[74] Franz Rothlauf. *Design of modern heuristics: principles and application*. Springer Science & Business Media, 2011.

[75] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010. ISBN 9780136042594.

[76] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 357–362. AAAI Press, 2008. ISBN 978-1-57735-368-3.

[77] Jendrik Seipp and Gabriele Röger. Fast downward stone soup 2018. *IPC2018–Classical Tracks*, pages 72–74, 2018.

[78] Jendrik Seipp, Thomas Keller, and Malte Helmert. Saturated cost partitioning for optimal classical planning. *Journal of Artificial Intelligence Research*, 67, 01 2020. doi: 10.1613/jair.1.11673.

[79] Bart Selman. Near-optimal plans, tractability, and reactivity. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, KR'94, page 521–529, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 155860328X.

[80] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30 (1):574–584, Jun. 2020. URL `https://ojs.aaai.org/index.php/ICAPS/article/view/6754`.

[81] Silvan Sievers, Michael Katz, Shirin Sohrabi, Horst Samulowitz, and Patrick Ferber. Deep learning for cost-optimal planning: Task-dependent planner selection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33 (01):7715–7723, Jul. 2019. doi: 10.1609/aaai.v33i01.33017715. URL `https://ojs.aaai.org/index.php/AAAI/article/view/4767`.

[82] Silvan Sievers, Daniel Gnad, and Álvaro Torralba. Additive pattern databases for decoupled search. In Lukás Chrpa and Alessandro Saetti, editors, *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, pages 180–189. AAAI Press, 2022. URL `https://ojs.aaai.org/index.php/SOCS/article/view/21766`.

[83] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, L. Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140 – 1144, 2018.

[84] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artif. Intell.*, 125 (1–2):119–153, January 2001. ISSN 0004-3702. doi: 10.1016/S0004-3702(00)00079-5. URL https://doi.org/10.1016/S0004-3702(00)00079-5.

[85] Roni Stern, Ariel Felner, and Robert Holte. Probably approximately correct heuristic search. *Proceedings of the 4th Annual Symposium on Combinatorial Search, SoCS 2011*, 01 2011.

[86] Roni Stern, Ariel Felner, and Robert C. Holte. Search-aware conditions for probably approximately correct heuristic search. In Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors, *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press, 2012. URL http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5374.

[87] Roni Stern, Ariel Felner, Jur van den Berg, Rami Puzis, Rajat Shah, and Ken Goldberg. Potential-based bounded-cost search and anytime non-parametric a∗. *Artificial Intelligence*, 214:1–25, 2014. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2014.05.002. URL https://www.sciencedirect.com/science/article/pii/S0004370214000551.

[88] Roni Stern, Gal Dreiman, and Richard Valenzano. Probably bounded suboptimal heuristic search. *Artificial Intelligence*, 267:39–57, 2019. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2018.08.005. URL https://www.sciencedirect.com/science/article/pii/S0004370218306015.

[89] S Suganyadevi, V Seethalakshmi, and K Balasamy. A review on deep learning in medical image analysis. *International Journal of Multimedia Information Retrieval*, 11(1):19–38, 2022.

[90] Tian-Xiang Sun, Xiang-Yang Liu, Xi-Peng Qiu, and Xuan-Jing Huang. Paradigm shift in natural language processing. *Machine Intelligence Research*, 19(3):169–183, 2022.

[91] Takeshi Takahashi, He Sun, Dong Tian, and Yebin Wang. Learning heuristic functions for mobile robot path planning using deep neural networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 764–772, 2019.

[92] Jordan Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI International Joint Conference on Artificial Intelligence*, pages 674–679, 01 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-119.

[93] Jordan Thayer, Austin Dionne, and Wheeler Ruml. Learning inadmissible heuristics during search. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2011.

[94] D. Toropila, F. Dvořák, O. Trunda, M. Hanes, and R. Barták. Three approaches to solve the petrobras challenge: Exploiting planning techniques for

solving real-life logistics problems. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012*, volume 1, pages 191–198. IEEE, 2012. ISBN 978-1-4799-0227-9.

[95] Sam Toyer, Sylvie Thiébaux, Felipe Trevizan, and Lexing Xie. Asnets: Deep learning for generalised planning. *Journal of Artificial Intelligence Research*, 68:1–68, 2020.

[96] Julia Wichlacz, Daniel Höller, and Jörg Hoffmann. Landmark heuristics for lifted classical planning. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 4665–4671. ijcai.org, 2022. doi: 10.24963/ijcai.2022/647. URL `https://doi.org/10.24963/ijcai.2022/647`.

[97] Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9(Apr): 683–718, 2008.

List of Figures

List of Tables

List of Abbreviations

- *DS* - Data structure.
- *EOG* - Extended object graph. See section 4.5.1.
- *FF* - FastForward (planning system). See section 1.1.5.
- *FDR* - Finite Domain Representation. See section 1.1.2
- *GA* - Genetic algorithm.
- *GRU* - Gated recurrent unit. See section 1.2.4.
- *HL* - Heuristic learning. See section 1.3.
- *IPC* - International Planning Competition.
- *LogMSE* - Logarithmic Mean Squared Error. See section 5.5.
- *LSTM* - Long short term memory. See section 1.2.4.
- *ML* - Machine learning.
- *MSE* - Mean Squared Error. See section 1.2.3.
- *MutEx* - Mutually Exclusive predicates. See definition 8.
- *NN* - (Artificial) neural network.
- *PDDL* - Planning Domain Description Language. See section 1.1.3.
- *PDB* - Pattern Database. See section 3.2.2.
- *RL* - Reinforcement learning.
- *RV* - Random variable.
- *SAS* - Simplified Action Structures. See section 1.1.3.
- *STRIPS* - Stanford Research Institute Problem Solver. See section 1.1.1.
- *TSP* - Travelling salesman problem.

List of publications

2021

Otakar Trunda and Roman Barták. Heuristic learning in domain-independent planning: Theoretical analysis and experimental evaluation. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Agents and Artificial Intelligence*, volume 12613 of *Lecture Notes in Computer Science*, pages 254–279. Springer International Publishing, 2021.

2020

Otakar Trunda and Roman Barták. Deep learning of heuristics for domain-independent planning. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of the 12th International Conference on Agents and Artificial Intelligence, ICAART 2020, Volume 2, Valletta, Malta, February 22-24, 2020*, pages 79–88. SCITEPRESS, 2020.

2017

Robert Brunetto and Otakar Trunda. Deep heuristic-learning in the rubik's cube domain: An experimental evaluation. In Jaroslava Hlaváčová, editor, *Proceedings of the 17th Conference on Information Technologies - Applications and Theory (ITAT 2017), Martinské hole, Slovakia, September 22-26, 2017*, volume 1885 of *CEUR Workshop Proceedings*, pages 57–64. CEUR-WS.org, 2017.

2016

Otakar Trunda. Monte carlo tree search as a hyper-heuristic framework for classical planning. In M. Katz N. Lipovetzky Ch. Muise M. Ramırez A. Torralba J. Benton, D. Bryce, editor, *Proceedings of the 8th Workshop on Heuristics and Search for Domain-independent Planning co-located with 26th International Conference on Automated Planning and Scheduling (HSDIP@ICAPS) 2016, London, UK, June 13, 2016*, pages 111–119. CreateSpace Independent Publishing Platform, 2016.

Otakar Trunda and Robert Brunetto. Fitness landscape analysis of hyper-heuristic transforms for the vertex cover problem. In Brona Brejová, editor, *Proceedings of the 16th ITAT Conference Information Technologies - Applications and Theory, Tatranské Matliare, Slovakia, September 15-19, 2016*, volume 1649 of *CEUR Workshop Proceedings*, pages 179–186. CEUR-WS.org, 2016.

2015

Tomáš Balyo, Roman Barták, and Otakar Trunda. Reinforced encoding for planning as SAT. *Acta Polytechnica CTU Proceedings*, 2(2):1–7, 2015.

2014

Otakar Trunda. Automatic creation of pattern databases in planning. In V. Kůrková, L. Bajer, M. Holeňa, M. Nehéz, L. Peška, and P. Vojtáš, editors, *Proceedings of the 14th conference ITAT 2014 – Workshops and Posters*, pages 85–92. Institute of Computer Science AS CR, 2014.

Otakar Trunda and Roman Barták. Determining a proper initial configuration of red-black planning by machine learning. In Joaquin Vanschoren, Pavel Brazdil, Carlos Soares, and Lars Kotthoff, editors, *Proceedings of the International Workshop on*

Meta-learning and Algorithm Selection co-located with 21st European Conference on Artificial Intelligence, MetaSel@ECAI 2014, Prague, Czech Republic, August 19, 2014, volume 1201 of *CEUR Workshop Proceedings*, pages 51–52. CEUR-WS.org, 2014.

2013

Otakar Trunda and Roman Barták. Using monte carlo tree search to solve planning problems in transportation domains. In Félix Castro-Espinoza, Alexander F. Gelbukh, and Miguel González-Mendoza, editors, *Advances in Soft Computing and Its Applications - 12th Mexican International Conference on Artificial Intelligence, MICAI 2013, Mexico City, Mexico, November 24-30, 2013, Proceedings, Part II*, volume 8266 of *Lecture Notes in Computer Science*, pages 435–449. Springer International Publishing, 2013.

2012

Daniel Toropila, Filip Dvorak, Otakar Trunda, Martin Hanes, and Roman Barták. Three approaches to solve the petrobras challenge: Exploiting planning techniques for solving real-life logistics problems. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 191–198. IEEE Computer Society, 2012.

A. Attachments

A.1 Description of planning domains used in the experiments

A.1.1 Zenotravel

Zenotravel describes a transportation problem. There are cities, persons and planes; persons and planes are located at cities. Goal is to transport persons and planes to their destinations. Planes can fly between any two cities and persons can board planes that are in the same city and they can leave the plane at any city.

Planes have unlimited capacity but limited amount of fuel. When flying from one city to another, one unit of fuel is consumed and when the fuel level reaches zero, the plane can't fly. At an time a plane can increase its fuel level by one unit by performing the *refuel* action. The task is to find a plan that minimizes the number of actions.

Finding optimal plans for Zenotravel is *NP-hard*. There is a 2-approximation algorithm but a polynomial-time approximation scheme does not exist unless *P=NP*. See [42], theorem 5.4.2.

PDDL files for the *Zenotravel* domain and 20 problems can be found on the *http://planning.domains* website, specifically on
http://api.planning.domains/json/classical/problems/17. Both PDDL and SAS versions of the problems we used in the experiments are provided in the supplementary materials.

A.1.2 Blocks

Blocks domain features a scenario with several blocks (containers) stacked on top of each other and a hoist to manipulate them. Goal is to rearrange the blocks from given initial configuration to a specified target configuration using as few actions as possible.

Block can be located either on the ground or on top of another block. Blocks that have no other block located on top of them are called *free*. Only free blocks can be moved by the hoist and only one block can be picked up at any time. When a block is picked up, it can be dropped down either on the ground or on top of another block that is currently free. No other actions are available.

Similar to the *Zenotravel*, computational complexity of *Blocks* is *NP-hard*, a 2-approximation algorithm exists but not a polynomial-time approximation scheme. See [79, 84].

PDDL files can again be found on the *http://planning.domains* website:
http://api.planning.domains/json/classical/problems/112. Both PDDL and SAS versions of the problems we used in the experiments are provided in the supplementary materials.

A.2 Description of ad-hoc solvers

We provide a brief description of ad-hoc solvers that we've developed to generate training data.

A.2.1 Zenotravel Solver

Our Zenotravel solver combines genetic algorithm (GA) with a greedy search to find high-quality solution for any Zenotravel problem in a reasonable time. We use a two-phase procedure. First, we split the problem into several disjoint components and then solve each component by a simple greedy algorithm. Given a Zenotravel problem P, let C be a set of all cities, M set of all planes and T set of all persons. We split P into a set of components Q such that every component $q_i \in Q$ contains a single plane and a set of persons. The number of components is the same as the number of planes. Components are disjoint, i.e., every plane is present in exactly one component and the same holds for every person. Formally: $Q = \{q_i \mid q_i = (m_i, T_i), T_i \subset T, i \neq j \Rightarrow (m_i \neq m_j, T_i \cap T_j = \emptyset), \bigcup_i \{m_i\} = M, \bigcup_i T_i = T\}$. Every component represents a sub-problem: the task in q_i is to transport persons T_i to their destinations using only a single plane m_i.

Every component $q_i = (m_i, T_i)$ can be solved by the following greedy procedure. First we create a *Delivery graph* (see [42]) of q_i (denoted D_i) whose vertices are cities and there is an edge from c_1 to c_2 if there exists a person $t \in T_i$ whose current location is c_1 and target location is c_2. We remove all isolated vertices from D_i and then we find its topological order.

Since planes have infinite capacity, component q_i can be solved by visiting vertices of D_i in a topological order. At every city c, the plane m_i loads all persons from T_i that are not yet at their destinations and unloads persons whose destination is c. Original and target location of the plane are added the beginning and at the end of the journey respectively.

We run GA to find the best possible splitting into components. We use a straightforward encoding: genome is a vector v of size $|T|$, where every $v[i]$ is assigned value from $\{1, 2, \ldots |M|\}$. $v[i] = j$ means that person t_i belongs to component q_j. The number of components is the same as the number of planes and every plane m_j belongs to component q_j. For every candidate solution v, we perform the splitting according to v, solve each component by the greedy algorithm and use sum of lengths of plans as the objective function. GA searches for splitting that minimizes the objective function.

Parameters of the GA are as follows: size of the population: 100, termination criterion: 20 generations passed without improving the solution. We use single point cross-over and several types of mutations.

A.2.2 Blocks Solver

Our blocks solver uses a simple greedy algorithm to find close-to-optimal solutions to blocks problems of any size. We utilize the notions of *GoodTower* and *BadTower* (see [72] for example). A block being *on table* means that there are no blocks under it. For a block X we say that *GoodTower(X)* is true if either of these conditions hold:

1. X is currently on table and goal position of X is on table or it is not defined
2. X is located on Y, goal position of X is on Y and *GoodTower(Y)* is true

Otherwise we say that the block is *BadTower*. Blocks that are *BadTower* have to be moved at least once. On the contrary, blocks that are *GoodTower* don't have to be moved and therefore should not be moved. Before we describe the algorithm, lets first define a few notions:

- let C be the set of blocks that are currently clear - i.e., there is no block located on top of them
- C_G be a set of blocks from C that are *GoodTower*
- C_B be a set of blocks from C that are *BadTower*
- $u(X)$ be the *unavailability* of a block X defined as the number of blocks that are located on top of X (height of the tower standing on top of X)
- GTC (good-tower candidates) be a set of blocks X such that there exists $Y_X \in C_G$ and goal position of X is on Y_X, together with blocks B whose goal position is on table or undefined. These are blocks that can be used to extend or start a good tower.

Our algorithm repeatedly applies the following rules: (the first applicable rule is used)

1. if there is $X \in C_B$ whose goal location is on table, move it to table
2. if there are $X \in C_B$ and $Y \in C_G$ such that goal location of X is on Y, move X on Y.
3. if $|GTC| \geq 1$ select $X \in GTC$ whose $u(X)$ is minimal, move all blocks on top of X to table (separately, not on top of each other), then move X such that *GoodTower(X)* is true
4. select block $X \in C_B$ randomly and move it to table

Steps *1* and *2* are special cases of step *3*.

A.2.3 *planning.domains* benchmarks

The *http://planning.domains* website stores more information about the benchmark problems than just their PDDL files. For each problem they keep track of the best plan found so far by anyone and they present these results in a form of lower bound and upper bound on the length of the optimal plan. These can be used by anyone to compare their solutions to optimal ones or at least to the best known ones.

If an optimal plan has been found (by an algorithm that guarantees optimality), both lower and upper bounds equal the length of the optimal plan. This is the case for some small problems but for the larger ones, algorithms that guarantee optimality are too slow to provide solution.

Lower bounds are produced by running an optimal algorithm, e.g. some sort of Breadth-First Search, for a limited time and then examining how far from the initial state the algorithm was able to reach. If it expanded some node as far as r steps from the initial state and still did not find a plan, then this can be used to guarantee that there is no plan shorter than r.

Upper bounds can be produced by providing a plan that solves the given problem. Validity of such plan can be checked quickly and length of the plan constitutes an upper bound on the length of the optimal plan.

The website provides an API and encourages users to submit their plans in order to improve the bounds. Submitted plans are validated automatically by the server before the bounds are updated.

We used our solvers to find solution for several of the problems for which optimal plans are not yet known and we have been able to improve the best known upper bounds for many problems. See
http://api.planning.domains/json/classical/problems/17 and
http://api.planning.domains/json/classical/problems/112
(the field *upper_bound_description*).

A.3 Content of attached archive

Domain files

Contains input files of the planning problems used in experiments. There are both PDDL and SAS versions. PDDL files were downloaded from
http://api.planning.domains/json/classical/problems/17 (zenotravel) and
http://api.planning.domains/json/classical/problems/112 (blocks). SAS variants were obtained using the translator available at
https://github.com/aibasel/downward/tree/main/src/translate.

Training data

All data samples we've collected. Folder states for training contains samples for individual problems. The structure is following: each row represents a state, columns:

- column 1: full ID of the state
- column 2: real goal distance of the state
- column 3: value of *FastForward* heuristic on that state
- columns 4-5 are redundant and used just to check consistency of our methods

Full ID of a state consists of domain name_file name_array of SAS state values. It uniquely identifies the state and the state can be reconstructed out of it.

The file *graphs.xlsx* contains statistical properties of the data, tables and graphs.

Heuristic accuracy

Accuracy of *FastForward* heuristic on the collected states. There is the raw data as well as various statistics and graphs.

Heuristic adjustments results

Search results for *FastForward* heuristic, *GoalCount* heuristic and their adjusted versions on both domains. Folder *results_30mins_1M* contains runs with time limit of 30 minutes and memory limit of 1 million expanded nodes. In folder *results_30mins_5M* there are runs with 5 million memory limit. Only results from *results_30mins_5M* are presented in the thesis.

Experiments results

Contains raw data of results of search runs. *FastForward* heuristic and the 12 variants of learned heuristics. There are aggregated results, tables and graphs.

source

Sources of our planner and our HL framework.

thesis.pdf

Text of this thesis.