Czech Technical University
Faculty of Electrical Engineering

Department of Computer Graphics & Interaction
Study major: Open Informatics

Optimizing portfolio heuristics for optimal planning

BACHELOR THESIS

Author: Daniel Žampach
Supervisor: Tomáš Pevný
Year: 2023/2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Žampach Daniel** Personal ID number: **508472**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Games and Graphics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Optimizing portfolio heuristics for optimal planning

Bachelor’s thesis title in Czech:

Optimalizace portoflia heuristik pro optimalni planovani

Guidelines:

This work is the concerned with optimizing portfolio of admissible heuristics for A* algorithm. By virtue of construction, the resulting heuristic is therefore guaranteed to be admissible, but the improvement is bounded by the quality of the heuristics in the portfolio. Hence this work will serve as a proof of concept of the idea and to study the properties of the approach.
1. Study the library NeuroPlanner.jl which will be used as a test bench.
2. Implement the portfolio planner in the library NeuroPlanner.jl. Test it on an ideal case, where the portfolio contains null and optimal heuristic by optimizing L* loss function.
3. Implement admissible heuristic lm-cut to the SymbolicPlanners.jl as a representant of the strong admissible heuristic.
4. Test the porfolio on at least three benchmark problems (e.g. those used in the paper "Optimize Planning Heuristics to Rank, not to Estimate Cost-to-Goal. Measure the number of expanded states and compare it to that of individual heuristics in the portfolio.

Bibliography / sources:

Learning Generalized Policies Without Supervision Using GNNs, Simon Stahlberg, Blai Bonet, Hector Geffner, 2022
Asnets: Deep learning for generalised planning, S Toyer, S Thiébaux, F Trevizan, L Xie, 2022
GOOSE: Learning Domain-Independent Heuristics, Dillon Ze Chen and Sylvie Thiebaux and Felipe Trevizan, 2023
Chrestien, Leah, et al. "Optimize Planning Heuristics to Rank, not to Estimate Cost-to-Goal." Advances in Neural Information Processing Systems 36 (2024).
Neuroplanner.jl: https://github.com/pevnak/NeuroPlanner.jl
SymbolicPlanners.jl
SymbolicPlanners.jl: https://github.com/JuliaPlanners/SymbolicPlanners.jl

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Tomáš Pevný, Ph.D. Artificial Intelligence Center FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **19.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **15.02.2026**

doc. Ing. Tomáš Pevný, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses

Praha, ...

.....
Daniel Žampach

Acknowledgement

I would like to thank my supervisor doc. Ing. Tomáš Pevný, Ph.D. without whom this work would not have been possible. I would also like to thank Ing. Michaela Urbanovská, who helped with this work many times. Lastly I would like to thank my family who supported me during the making of this work.

Daniel Žampach

Title:
Optimizing portfolio heuristics for optimal planning

Author: Daniel Žampach

Study major: Open Informatics
Type: Bachelor thesis

Supervisor: Tomáš Pevný
Czech Technical University

Abstract: Finding an optimal plan (solution) to a planning problem can be a task unsolvable in a reasonable time. So it is important to efficiently comb the state-space of the problem. One viable option is the A^* planner the efficiency of which is heavily reliant on the heuristic function used, it should both give good information and be admissible. This work is focused on the finding of these heuristics. This is achieved by training a neural network to output a series of numbers to be used as convex weights for already existing admissible planning heuristics. The heuristic created as a convex combination of admissible heuristics is also an admissible heuristic, and when used with the A^* planner an optimal solution is guaranteed to be found. The heuristic is focused on lowering the number of expanded states by the A^* planner, and therefore lowering the overall time of plan finding. This work discusses the advantages and disadvantages of this approach as well as the fundamental limit on performance.
Keywords: planning, neural networks, admissible heuristics

Název práce:
Optimalizace portoflia heuristik pro optimalni planovani

Autor: Daniel Žampach

Abstrakt: Nalezení optimálního řešení plánovacího problému může trvat neúměrně dlouho. Z tohoto důvodu je potřeba prohledávat stavový prostor problému efektivně. Jednou z osvědčených možností je A^* plánovač, jehož účinnost je přímo závislá na použité heuristické funkci. Tato funkce by měla být přípustná a zároveň poskytovat užitečné informace. Tato práce se věnuje hledání těchto heuristik. Heuristiky nacházíme pomocí trénování neuronové sítě tak, aby produkovala sérii čísel. Ty jsou poté použity jako konvexní koeficienty pro již existující přípustné heuristiky. Heuristika, která vznikne jako konvexní kombinace přípustných heuristik, je také přípustnou heuristikou, a když ji použijeme s A^* plánovačem, je zaručené nalezení optimálního řešení. Naše heuristika se zaměřuje na zmenšení počtu prozkoumaných stavů A^* plánovačem, a tedy celkově zkrácením času potřebného k nalezení plánu. Tato práce diskutuje výhody a nevýhody tohoto přístupu a zároveň jeho fundamentální hranici výkonu.
Klíčová slova: plánování, neuronové sítě, přípustné heuristiky

Contents

List of Figures

Chapter 1

Introduction

Planning problems find use in a wide variety of fields. They can be thought of as state-searching problems, having defined sets of states, and transitions between them. Solving them can be done using state-searching algorithms. One of those the A^* algorithm is an effective and popular choice. It, however, needs a heuristic function to work, and its performance depends heavily on this heuristic function.

Finding a good heuristic function for planning problems can be difficult, as planning domains vary in structure and the problems themselves vary in complexity. As such recent approaches have had success by obtaining them by training a neural network to simulate a heuristic function from already solved problems.

NeuroPlanner.jl [1] is a Julia library which takes this approach. It provides several different representations of the planning problem, as well as various loss functions to help optimize the parameters of these heuristic functions. The heuristic created by NeuroPlanner is not admissible. This means that when using it with A^* we do not get the optimality guarantee (2.2.1).

This work is an expansion of the NeuroPlanner library, creating a new admissible heuristic h_{ANH} on top of the existing core functionalities. The h_{ANH} heuristic is created as a convex combination of several pre-existing admissible heuristics. It is then optimised to rank, in an attempt to lower the number of states expanded during the A^* algorithm as much as possible.

Chapter 2

Background in planning

We begin with the definitions and descriptions necessary for our work. This section covers how planning problems look, how they are commonly solved and what we use to solve them.

2.1 Planning tasks

A classical planning task (problem) is a tuple

$$\Pi = (\mathcal{S}, s_{\text{init}}, S_{\text{goal}}, A, f) \tag{2.1}$$

where:

- $\mathcal{S}$ is a discrete finite set of possible states
- $s_{\text{init}} \in \mathcal{S}$ is the initial state
- $S_{\text{goal}} \subset \mathcal{S}$ is a set of goal states
- $\mathcal{A}$ is a set of actions
- f is a function mapping actions taken in a state to a new state

$$s' = f(s, a) \tag{2.2}$$

This definition was taken from [2].

For certain processes (such as calculating LM-Cut later on), it is beneficial to reformulate the planning task in the STRIPS format. A STRIPS planning task is a tuple where:

$$\Pi = (\mathcal{F}, O, s_{\text{init}}, s_{\text{goal}}, c) \tag{2.3}$$

- $\mathcal{F}$ is a finite set of facts
- $\mathcal{O}$ is a set of operators, which each are a triplet

$$\text{o} \in \mathcal{O} = (pre(\text{o}), add(\text{o}), del(\text{o}) \tag{2.4}$$

- $s_{\text{init}} \subset \mathcal{F}$ is the initial state
- $s_{\text{goal}} \subset \mathcal{F}$ is a goal specification
- c is the cost function of actions, $c : \mathcal{O} \longrightarrow \mathbb{R}$

In this definition, a state is any subset $\text{s} \subset \mathcal{F}$. Further operators work in the manner that operator o can only be used in state s if its preconditions are satisfied in that state, i.e. $pre(\text{o}) \subset \text{s}$. The effect of the operator is as such: $\text{s}' = \text{s} \cup add(\text{o}) - del(\text{o})$. Operators are also sometimes called *actions*, in line with the classical task definition.

A fact can also be represented as a combination of predicates and objects. The objects are a list of named variables that exist in the problem. The predicates are operators on these objects, they take an input list of objects of length m and have a binary output $\{0, 1\}$, where a 1 represents

that the predicate with those objects is a fact present in the given state. The length m is in the range $\langle 0, n \rangle$, where n is the number of objects in the problem. This means the predicates can also take no objects as input (nullary predicates). This allows us to give the facts an internal structure and establish relationships between them. A fact is also commonly referred to as an *atom*.

While we are defining planning tasks it is important to mention that they are usually not taken as individual problems. They belong to domains, which are groups of problems with some common logic and rules. These domains usually define how the states and actions look in classical planning and how the different facts and operators interact in STRIPS. To understand a planning task fully, we need both the domain definition and the problem definition. These are usually written in a language called PDDL, which was created for this purpose.

Solving a planning problem means finding a plan. A plan is a sequence of actions (or operators in STRIPS) which will when applied sequentially to the initial state, output the goal state of the problem. Formally (for classical planning): A plan $\pi = (a_1, ..., a_n)$ is a sequence of n actions $a \in A$ such that the following holds:

$$f(s_{\text{init}}, a_1) = s_1 \tag{2.5}$$

$$s_{i+1} = f(s_i, a_{i+1}) \tag{2.6}$$

$$s_n \in S_{\text{goal}} \tag{2.7}$$

An optimal plan is then a plan whose cumulative cost of actions $c_{sum}(\pi) = \sum_{i=1}^{n} c(a_i); a_i \in \pi$ is the lowest of any plan $\pi \in \Pi$ where Π is the set of all plans for the given problem.

$$\pi_{opt} = \arg\min_{\pi \in \Pi} c_{sum}(\pi) \tag{2.8}$$

Finally, an optimal path is a sequence of states we get if we apply the actions from an optimal plan sequentially to the initial state. We will note this as S^{π}.

2.2 Solving planning tasks

To solve planning tasks in a domain-independent way, state-searching algorithms are used. These are algorithms such as the breadth-first search, greedy search etc.. However, in this work, we focus on the A^* algorithm. The A^* algorithm is an algorithm for reaching goal states by searching the state space and returning the plan needed to reach them. A^* by always expands the state with the lowest value of the merit function:

$$f_{A^*}(s) = g(s) + h(s) \tag{2.9}$$

Here, $g(s)$ is the distance function which returns the cost of reaching the given state from the initial state (or the distance to the initial state in uniform cost functions). $h(s)$ is then any heuristic function, which must be provided for the algorithm to function. Formally:

Algorithm 1 The A^* algorithm

Require: A heuristic function h(s)
  $Q \leftarrow \{s_{init}\}$
  $V \leftarrow \{s_{init}\}$
  while $Q \cap S_{goal} = \emptyset$ **do**
    $s \leftarrow \arg\min_{q \in Q} f_{A^*}(q)$
    $E \leftarrow \{s' | \exists a \in A; s' = f_{transition}(s, a); s' \notin V\}$
    $Q \leftarrow Q \cup E - s$
    $V \leftarrow V \cup s$
  end while

Note that since both the transition function from planning task definition, and the merit function from A^* are traditionally called f, we label them $f_{transition}$ and f_{A^*} to differentiate. The A^* algorithm is most commonly implemented using a priority queue (to get the argument minimum quickly), where the priority is the merit function f_{A^*}, as such the merit function is also sometimes referred to as *priority*.

The reason we focus on A^* is the following property:

Lemma 2.2.1 (A^* guarantee)**.** If the heuristic used by the A^* algorithm is *admissible*, then A^* is guaranteed to find an optimal plan.

This property is important, as it guarantees that any plan we find using the A^* algorithm is going to be optimal. Other common state-searching algorithms do not have this guarantee and, therefore, we can not be sure if a shorter plan exists or not.

A heuristic function is a function mapping states to real numbers. It is used to indicate inverse priority of states during search, the states with the lowest heuristic function value should be searched first.

$$h(s) : S \longrightarrow \mathbb{R} \tag{2.10}$$

A heuristic is admissible if it never overestimates the cost to goal and it is never negative. If the optimal (true) cost to goal function is $h_*(s)$, then a heuristic function h is admissible if:

$$\forall s \in S; 0 \leq h(s) \leq h_*(s) \tag{2.11}$$

Lemma 2.2.2 (Convex combination of admissible heuristics)**.** Given a series of admissible heuristic functions h_1, ..., h_n, any convex combination of these heuristics is again an admissible heuristic function. Formally if

$$h' = \alpha_1 * h_1 + \alpha_2 * h_2 + ... + \alpha_n * h_n \tag{2.12}$$

subject to:

$$\sum \alpha_i = 1; 0 \leq \alpha_i \leq 1; \forall i \in \{1, 2, ...n\} \tag{2.13}$$

Then h' is admissible.

Proof: The minimum value the h' heuristic can achieve is the value of the heuristic h_i which has the minimal value for the given state among all of the heuristics $h_i; i \in \{1, 2, ..., n\}$. This value is achieved when the corresponding coefficient α is equal to one, $\alpha_i = 1$, and the other coefficients are equal to zero, $\alpha_j = 0; j \neq i$. Since this minimal heuristic h_i is admissible, then the minimal values must be greater or equal to zero, $h_i(s) \geq 0; \forall s \in \mathcal{S}$, and thus the h' heuristic has the same lower bound, $h'(s) \geq 0; \forall s \in \mathcal{S}$ as its minimal value is $h' = 1 * h_i$. In the same way, there is a maximal heuristic h_l which has the greatest heuristic value among all the heuristics $h_l; l \in \{1, 2, ..., n\}$, and the maximal value of h' is reached when the α coefficient of h_l is one $\alpha_l = 1$ and the other coefficients are zero $\alpha_k = 0; k \neq l$. Since h_l is admissible the maximum value it can have for any state is equal to the true cost-to-goal h_*, $h_l(s) \leq h_*(s); \forall s \in \mathcal{S}$. The maximal value is then $h' = 1 * h_l$, meaning h' has an upper bound of h_*, meaning: $h'(s) \leq h_*(s); \forall s \in \mathcal{S}$. As such h' is admissible as it never overestimates the true cost-to-goal and is never negative, $0 \leq h'(s) \leq h_*(s); \forall s \in \mathcal{S}$.

2.3 Common admissible heuristics

Thanks to the A^* guarantee 2.2.1, admissible heuristics are desirable in the solving of planning problems. Of the many that exist we will only focus on a select few which are commonly used. The zero heuristic h_0 (the null heuristic), is a heuristic which assigns a 0 to every state.

$$h_0(s) = 0; \forall s \in \mathcal{S} \tag{2.14}$$

Since the cost to goal in traditional planning can not be negative, the h_0 heuristic never overestimates and is thus admissible.

2.3.1 $h_{\max}$

The $h_{\max}$ [3] heuristic is a member of the deletion-relaxation family of heuristics. These heuristics seek to estimate the true cost to goal, by finding the cost to goal of a simplified variant of the problem. $h_{\max}$ works with the STRIPS definition of planning tasks and it simplifies the problem by removing the deletion effect of operators:

$$\forall \mathrm{o} \in \mathcal{O} : \mathrm{o} = (pre(\mathrm{o}), add(\mathrm{o}), del(\mathrm{o}) \longrightarrow \mathrm{o} = (pre(\mathrm{o}), add(\mathrm{o}), \emptyset) \tag{2.15}$$

$h_{\max}$ then takes this simplified problem and constructs a relaxed-planning graph. This graph has vertices represented by facts and edges by the altered operators. For our purposes, we can disregard the edges and simply imagine the graph as a collection of facts $\mathcal{F}_{\text{rpg}}$ that each fact is either in or not.

Its creation starts by taking the facts in the initial state $\mathcal{F}_{\text{rpg}} = \mathcal{I}$ and then works in iterations, always adding a fact $f \in \mathcal{F}$ which is reachable from the current facts, and of those reachable states it the operator that add it has the lowest cost, $\text{o} = \arg\min c(\text{o})$. Reachable here means that all of the preconditions of an action that has the given fact as one of its add effects are already in the planning graph, $pre(\text{o}) \subset \mathcal{F}_{\text{rpg}}, f \in add(\text{o})$. We also assign each fact a cost when we add it to the graph, this is equal to the cost of the operator which adds it plus the cost of the cheapest precondition of that operator. Let us call the function performing this mapping δ, $\delta(f) = c(o) + \min_{f_{pre} \in pre(\text{o})} \delta(f_{pre}); f \in add(\text{o})$. The iterations end when all the reachable facts have been added to the graph. This maps each fact in $\mathcal{F}$ to the cost of achieving it from the current state using the altered operators, i.e. what is the cumulative cost of operators which lead from the initial state to a state containing the given fact. The output value of the $h_{\max}$ heuristic is then the maximum of the costs of the facts in the goal state,

$$h_{\max}(s) = \max_{\text{f} \in s_{\text{goal}}} \delta(\text{f}); \tag{2.16}$$

The $h_{\max}$ heuristic is admissible, as even in the worst-case scenario in which removing deletion effects of operators did not simplify the problem, it only outputs the cost of reaching the goal.

2.3.2 $h_{\text{LM-Cut}}$

The $h_{\text{LM-Cut}}$ [4] heuristic is a heuristic from the landmark family of heuristics, they focus on finding landmarks, which are actions or facts of planning problems which must be included by every plan (or the path induced by the plan). In this work, we will be focusing on disjunctive action landmarks (further just landmarks), which are sets of actions one of which must be included in every plan. The $h_{\text{LM-Cut}}$ algorithm works to find these landmarks iteratively. It works by expanding $h_{\max}$, by relying on the values of the δ function from $h_{\max}$. In each iteration, it first calls the $h_{\max}$ algorithm and retrieves from it the δ function and the $h_{\max}$ heuristic value for that state. If the heuristic value is infinite, then the value of the $h_{\text{LM-Cut}}$ heuristic is also infinite and the algorithm terminates. Otherwise, it finds a supporter for each operator $\text{o} \in \mathcal{O}$, which is the fact from the preconditions of the operator with the highest cost in the δ function, let's call the function mapping operators to supporters supp:

$$\text{supp}(\text{o}) = \arg\max_{\text{f} \in pre(\text{o})} \delta(f) \tag{2.17}$$

Example Let's suppose we have a planning problem[1] that has the facts $\mathcal{F} = \{a, b, c, d, e, f, g\}$, the initial state $\mathcal{I} = \{a, c\}$, goal state $\mathcal{G} = \{a, g\}$ and operators as follows:

	pre(o)	add(o)	del(o)	c(o)
o_1	a	b	$\emptyset$	1
o_2	a,c	d,e	c	2
o_3	b,d	f	d	1
o_4	d,e	g	$\emptyset$	3
o_5	b,e	g	b	1

Then if we follow the $h_{\max}$ algorithm we will get the following values of the δ function:

	δ(f)
a	0
b	1
c	0
d	2
e	2
f	3
g	3

And as such the values of the supp function will be (ties are broken arbitrarily here):

	$supp$(o)
o_1	a
o_2	a
o_3	d
o_4	d
o_5	e

[1] This planning problem was provided by Ing. Michaela Urbanovská in private conversation.

With the supp function, the algorithm then creates a justification graph. The oriented graph has vertices represented by the facts $f \in \mathcal{F}$ and edges in the form of: $(\text{supp}(\text{o}), \text{f}), \text{f} \in add(\text{o})$. These edges are labelled by the corresponding operator o. New facts and operators are also added during the construction of the justification graph. First, a new initial fact $\mathcal{I}$, along with the operator $\text{o}_{\text{init}} = (\mathcal{I}, \{f; f \in s_{\text{init}}\}, \emptyset)$. Second, a new goal fact $\mathcal{G}$, with the operator $\text{o}_{\text{goal}} = (\{f; f \in s_{\text{goal}}\}, \mathcal{G}, \emptyset)$. Notably, the costs of both of the new operators are 0, $c(\text{o}_{\text{goal}}) = c(\text{o}_{\text{init}}) = 0$. New justification graph edges are then created from these new facts and operators just like with the previous edges and facts ($supp(\text{o}_{\text{init}}) = \mathcal{I}$, even though $\mathcal{I}$ does not have a defined δ value).

Example continued: Let us continue the last example. First, we expand the set of facts with $\{\mathcal{I}, \mathcal{G}\}$, $\mathcal{F} = \{a, b, c, d, e, f, g, \mathcal{I}, \mathcal{G}\}$. Second, we add the new operators and their supporters:

	pre(o)	add(o)	del(o)	c(o)	supp(o)
o_{init}	$\mathcal{I}$	a,c	$\emptyset$	0	$\mathcal{I}$
o_{goal}	a,g	$\mathcal{G}$	$\emptyset$	0	g

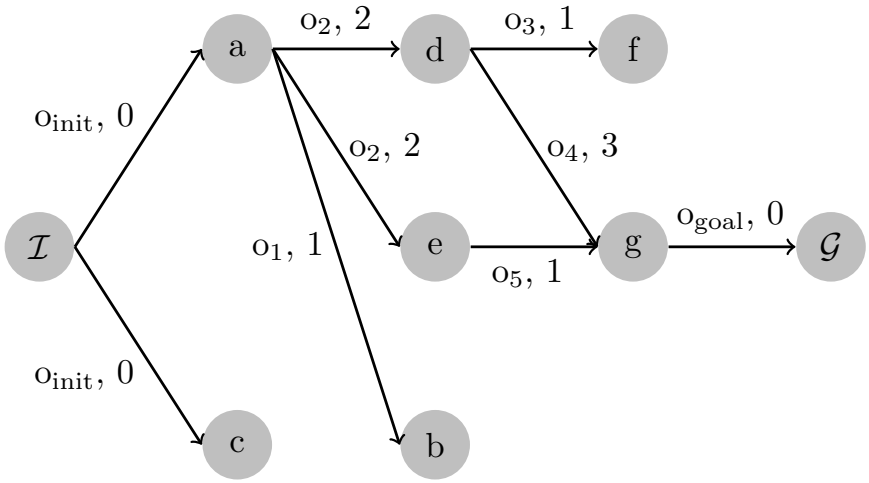

Figure 2.1: Example justification graph

Figure 2.1 shows the justification graph. The edges are labelled with a tuple of consisting of the name of the operand o the edge represents, and the cost of that operand c(o). A simple way of understanding the construction of the justification graph is that we first take all of the facts we are working with, $\mathcal{F}$, to form the vertices and then we connect them with edges as described. Since each fact in the add effect of each operator must be the end point of some edge (the justification contains *all* the edges fulfilling the described form), we can simply cycle through all operators and their add effects to generate the edges (see *Implementation* 4.2).

With the justification graph created, the algorithm then finds the N_* and N_0 partitions. The N_* partition is a set of all the vertices of the graph, which are reachable from the goal fact $\mathcal{G}$ using only edges whose operator (the label of the edge) has a cost of 0, $c(\text{o}) = 0$. The N_0 partition is then found as the set of all vertices reachable from the start fact $\mathcal{I}$ without using edges that contain a fact from the N_* partition. This separates the graph into two parts (the leftover states that belong neither in N_* nor N_0 are not used further in the algorithm). When an edge crosses between these two parts, meaning its starting fact is from the N_0 partition and the ending fact is from the N_0 partition, we call the operator labelling this edge a landmark. We then find the landmark with the lowest cost and call that cost σ. Finally, the cost of each landmark is lowered by σ, $\text{c}(\text{o}_{\text{lm}})_{\text{t}+1} = \text{c}(\text{o}_{\text{lm}})_{\text{t}} - \sigma$.

Example continued: Finding the N_* and N_0 for the example justification graph 2.1 is simple. There is only one fact reachable from $\mathcal{G}$ by using operators of cost 0, that being g, N_* is then $N_* = \{g, \mathcal{G}\}$. N_0 is then all of the other facts, as they are all reachable from $\mathcal{I}$ without passing through $\{g, \mathcal{G}\}$, i.e. $\mathcal{I} = \{a, b, c, d, e, f, \mathcal{I}\}$. Two edges start in N_0 and end in N_* - the edge labelled o_4 going from d to g, and the edge labelled o_5 going from e to g. This means that o_4 and o_5 are the landmark operators in this iteration. Of those, o_5 has the lowest cost at $c(\text{o}_5) = 1$, so we set $\sigma_1 = c(\text{o}_5) = 1$ and we lower the costs of the landmarks by it. After the first iteration the operators o_4 and o_5 now look like:

	pre(o)	add(o)	del(o)	c(o)
o_4	d,e	g	$\emptyset$	2
o_5	b,e	g	b	0

A new iteration then begins with the altered operator costs. The algorithm terminates when the cost of reaching the goal facts in the relaxed planning graph becomes zero. The output of the heuristic is then the sum of all σ across all (here N) iterations.

$$h_{\mathrm{LM-Cut}}(s) = \sum_{i=1}^{N} \sigma_i \tag{2.18}$$

Chapter 3

Implementing heuristics by neural networks

There are different ways how to design heuristic functions. Presently we know several major approaches how to design usable heuristics, those being: deletion relaxation, critical paths, abstractions, and landmarks [4]. Due to the recent successes, we are trying to create them by machine learning with neural networks.

3.1 Approaches with neural networks

The idea of using neural networks to find heuristics for classical planning is not new. The differing approaches vary primarily in the way the planning problem is represented. Each state within a planning problem needs to be transformed into a form suitable for the neural network. Toyer et al. [5] used Action Schema Networks which learn a common set of weights for all problems in a domain. Shen et al. [6] created a representation of the planning using hyper-graph neural networks to learn domain-independent heuristics. Chen et al. [7] expand on this method by creating three new graph representations for planning problems, that are created from the lifted representation of the problem. In general, the representations need to use the information about the state to make sure it is distinguishable from the rest. This commonly means to take the facts in the state and describe their inner structure - the predicates and the objects. The predicates and the objects are encoded into feature vectors for the neural networks to be able to parse them. There is not a standardized way of doing this and new methods are still being developed [8]. If the models give two different states an identical representation, then the model can not learn what the proper output for each should look like and this causes problems (see *Domain choice* 5.3.2).

These representations are then commonly used so that the given neural network may learn to produce a singular scalar output - the heuristic value. This does not result in an admissible heuristic, as the model trains, it is not bounded by the true cost to goal, and as such we can not guarantee that the output will not overestimate. This can be beneficial because the heuristic not having an upper bound can allow the model to learn to recognize states that are not likely to lead to the goal state quickly and assign them a high heuristic value. But if we want to take advantage of the benefits of admissible heuristics, like the A^* guarantee (2.2.1), then these methods are insufficient.

3.2 The admissible neuro-heuristic

The solution we propose is the admissible neuro-heuristic h_{ANH}, which is an admissible heuristic function realized by a graph neural network model. The heuristic value of this function for any given state is calculated as a convex combination of the values of several admissible heuristic functions for that state. Since they are admissible and the coefficients of the combinations are convex, then according to lemma (2.2.2), the new heuristic is guaranteed to be admissible.

$$h_{\mathrm{ANH}}(s) = \alpha_1 h_1(s) + \alpha_2 h_2(s) + ... + \alpha_{\mathrm{n}} h_n(s) \tag{3.1}$$

subject to:

$$\sum \alpha_{\mathrm{i}} = 1; 0 \leq \alpha_{\mathrm{i}} \leq 1; \forall i \in \{1, 2, ...n\} \tag{3.2}$$

The heuristic is constructed in the following way. First, we select n admissible heuristic functions $h_1, h_2, ..., h_n$. Which functions we choose should not theoretically matter, as any admissible heuristic should suffice, but as we will discuss later (in *Choosing input heuristics* 5.1.1), the performance of the individual heuristics affects the output noticeably. Secondly, we alter the final layer of the neural network model by increasing the dimension of the layer to n from 1 to match the number of heuristics. These output scalars will be the pre-normalization coefficients of the heuristics. Now we wrap the old final layer f_{ofl} of the model with the new final layer f_{nfl} which was created as follows:

$$(\alpha_1, \alpha_2, ..., \alpha_{\text{n}}) = softmax(\text{f}_{\text{ofl}}(\text{x})) \tag{3.3}$$

$$f_{nfl}(x) = \sum_{i=1}^{n} (\alpha_i h_i(s)) \tag{3.4}$$

Where x is the input data of the final layer. Put plainly, the new final layer first calls the old final layer to get the pre-normalization coefficients, then calls softmax on them, ensuring their convexity. Finally, we calculate the combination and this number is the output heuristic of the model. This evades the problem of the neural network output being unbounded. The softmax function creates this bound by itself, not allowing any of the coefficients α_i to be outside of the $< 0, 1 >$ range.

The notable advantage of the h_{ANH} heuristic is being able to easily take advantage of other admissible heuristics by simply adding it to the list of input heuristics. As the ways of designing heuristics are varied so are the performances in varying situations. The h_{ANH} heuristic can learn in which situations which of the input heuristics performs best and simulate the outputs of the heuristic for the given situation. It can find that a certain combination of the heuristics gives better information. The disadvantage is that the heuristic requires the evaluation of multiple heuristic functions per the evaluation of h_{ANH}. With simple heuristics this is negligible, but for more complex heuristic functions the computation times can rise sharply. A slow heuristic that gives good information might perform worse than a fast heuristic that gives bad information simply because it manages to expand orders of magnitude more states during the search.

3.3 Realizing the admissible neuro-heuristic

3.3.1 NeuroPlanner

The h_{ANH} is realized as an extension of the NeuroPlanner.jl [1] library. The library facilitates the creation of heuristic functions based on neural networks. It features implementations of the above-mentioned problem representations (ASNets, HyperGraphs... etc.) [6, 7, 5] . It creates an extractor that converts the PDDL problem into the corresponding representation. This extractor is domain-dependent, it receives the domain for which it functions during creation. This means the created h_{ANH} heuristic is also domain-dependent. It then uses the Mill.jl [9] library to produce a model compatible with the representation. This model is then trained using one of NeuroPlanner's implementations of loss functions and by using the Flux.jl library which provides the means for training the model. For our purposes we will be using the L^* loss function [10].

3.3.2 L^* loss function, optimizing to rank

The chosen loss function L^* is notable, as typically, neural network-based heuristic functions are optimized for cost-to-goal, i.e. approximating the true cost of the actions of the optimal plan. The loss function then penalizes distance from the true cost-to-goal. L^* on the other hand is one of a family of loss functions which focuses on optimizing to rank. This means that the L^* loss function tries to achieve a situation in which the states s on the optimal path have a lower $f_{A^*}(s)$ than those *adjacent*. This means that we are not attempting to get as close to the true cost-to-goal heuristic, and instead we are only attempting to ensure the presence of inequalities [10]:

$$f_{A^*}(s_i^{opt}) < f_{A^*}(s_j); \forall s_i^{opt} \in S^{\pi} \tag{3.5}$$

Where s_j is a state *adjacent* to the state on the optimal path s_i^{opt}. *Adjacent* in this sense means any state not from the optimal path which could be competing with s_i^{opt} for expansion.

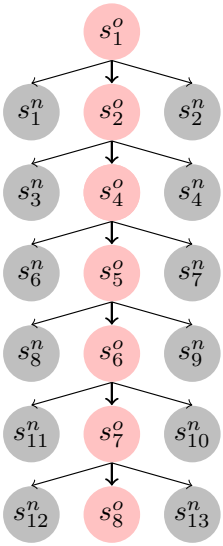

Figure 3.1: Adjacency example

For any given optimal state, this means not only non-optimal states sharing the same open state but also non-optimal states that were encountered earlier during the search. The figure (5.9) showcases an example of the search tree, s_1^o being the initial state and s_8^o being the goal state, the optimal path marked in red. In this case, the adjacent states to the optimal state s_4^o are not only non-optimal states s_3^n, s_4^n, but also states s_1^n, s_2^n.

L^* specifically then is a loss function that evaluates how close any given plan is to being optimal to rank. If $f_{A^*}(s_i^{opt}) = g(s_i^{opt}) + h(s_i^{opt})$ and $f_{A^*}(s_j) = g(s_j) + h(s_j)$, then the formulation of its condition on states is as such [10]:

$$r(s_i^{opt}, s_j) = g(s_i^{opt}) - g(s_j) + h(s_i^{opt}) - h(s_j) > 0 \tag{3.6}$$

The loss function then penalizes the breaking of these inequalities. Note that the original definition works with a heuristic function defined as $h(s, \theta)$, where θ are the function parameters.

3.3.3 Optimizing for fewest expanded states

The admissible heuristic has an advantage over conventional approaches in generating an admissible heuristic function; it is able to expand fewer states during planning with the A^* planner. This is given by two factors. The first factor, the L^* loss function mentioned above, allows us to avoid forcing the heuristic value to be close to the true cost-to-goal. This is beneficial as it means that states on the optimal path can have a much lower heuristic value than the true cost-to-goal, thus gaining higher priority and being expanded sooner. The second factor facilitates this: the list of input heuristics chosen will always include the zero heuristic h_0 (the null heuristic). This heuristic returns 0 for any given state, and therefore it is always admissible. The zero heuristic has numerous benefits, such as being extremely simple to calculate at any time and the aforementioned admissibility, the main reason for it being used here is its use as a way to lower the value of the convex combination. As described before, the weights (coefficients) of the input heuristics must sum to one $\sum \alpha_i = 1$. Since we are combining admissible heuristics, they can not be negative, and so if our list of heuristics does not include h_0 then the combination would not necessarily be able to become 0. The minimum possible value of h_{ANH} would be reached when $\alpha_i = 1; \alpha_j = 0; i \neq j$ where α_i is the alpha corresponding to input heuristic with the lowest value for the given state $h_{min} = \arg\min_{h \in H} h(s)$, where H is the list of all of the input heuristics of h_{ANH}. The minimum possible value over all the possible combinations of α_i coefficients would then be:

$$\min_{\alpha} h_{\text{ANH}}(s, \alpha) = h_{min}(s) \tag{3.7}$$

The h_0 heuristic allows us then to get the minimum value of the h_{ANH} to be 0, which is the minimal value of any admissible heuristic. This, in combination with L^*, boosts the capacity of the neural network model to push down the heuristic values on the optimal path. The A^* algorithm then expands these states faster, leading to fewer states being expanded. In a similar fashion L^* also drives up the heuristic values of non-optimal states as much as possible, giving them less priority and leading to them being expanded less often.

3.3.4 The upper limit of performance

Due to how the admissible neuro-heuristic h_{ANH} is constructed, we know the lower and upper bounds on its values. The lower bound is 0 as described above. We can similarly get the upper bound. We can also quickly set an upper bound from the fact that the h_{ANH} heuristic is admissible, and therefore it is bound by the true cost-to-goal h_*.

$$\max_{\alpha} h_{\mathrm{ANH}}(s, \alpha) = h_*(s) \tag{3.8}$$

We can lower this upper bound further, in a similar process as above, when $\alpha_i = 1$, which is the alpha corresponding to the input heuristic with maximum value for that state. $h_m = \arg\max_{h \in H} h(s)$, and $\alpha_j = 0; i \neq j$, where H is the list of input heuristics (we are naming the maximum input heuristic h_m instead of $h_{\max}$ to avoid confusion with the deletion relaxation heuristic $h_{\max}$). In other words, the maximum possible value the h_{ANH} heuristic can output for any state and any combination of α_i coefficients is the value of the maximum input heuristic h_m in that state:

$$\max_{\alpha} h_{\mathrm{ANH}}(s, \alpha) = h_m(s) \tag{3.9}$$

Thus the output range of h_{ANH} for any chosen heuristics and any state s of any planning problem is bound by $\langle 0; h_m(s) \rangle$.

$$h_{\mathrm{ANH}}(s) : S \longrightarrow I \subset \langle 0; h_*(s) \rangle; s \in S \tag{3.10}$$

The A^* algorithm does not only take into account the heuristic value, however. The g part of $f_{A^*}(s) = g(s) + h(s)$ is the function measuring the distance from the start state. The problem then arises, that we cannot control the value of g, it simply grows as we get further away from the initial state. This presents a problem, we only want A^* to expand the states on the optimal path, but since the path is optimal (meaning there is no shorter way to get to the goal and by extension, any of the states on the optimal path) the g function will grow linearly over its length. Naturally, non-optimal states close to the initial state will have a lower g than optimal states further away from it. This is the reason why A^* will expand a lot of non-optimal states even in situations with good heuristic functions. If we want to lower the number of states expanded, then we need to make the differences between optimal and non-optimal states in the value of the merit function $f_{A^*}(s)$ maximal possible. The heuristic function h chosen for A^* needs to compensate for the growth of the distance function g.

The ideal situation for our heuristic function h_{ANH} is:

$$h_{\mathrm{ANH}}(s) = 0; s \in S^{\pi} \tag{3.11}$$

$$h_{\mathrm{ANH}}(s) = h_m(s); s \in S - S^{\pi} \tag{3.12}$$

Where S^{π} is an optimal path of the problem.

As this is the best possible scenario, it also represents the fundamental upper limit on our approach. A^* with h_{ANH} will never expand less states than in this configuration. Therefore, these are the heuristic values we are attempting to achieve.

Chapter 4

Implementing LM-Cut

Because the implementation of the $h_{\mathrm{LM-Cut}}$ heuristic was key for this work (see *Choosing input heuristics* 5.1.1), and as the evaluation time of the implementation is important to the performance of the h_{ANH} heuristic, we would like to show how the $h_{\mathrm{LM-Cut}}$ heuristic was put into practice.

4.1 SymbolicPlanners

SymbolicPlanners.jl [11] is a Julia library that provides planners and heuristics for those planners, in order to solve planning problems. NeuroPlanner is made to function as a heuristic from SymbolicPlanners, the neuro-heuristic that is the output of NeuroPlanner is a type extension of the SymbolicPlanners heuristic type. This allows it to be easily slotted into any of the planners that SymbolicPlanners provides, most notably for us, the A^* planner. The A^* planner is a planner that uses the A^* algorithm to solve a given planning problem. It also provides information about the search, notably the success status, the search tree, the priority queue and the search frontier. The priority that the SymbolicPlanners A^* planner works with is not a scalar, but instead a tuple. This tuple has the form of $(f_{A^*}, h, size)$, where f_{A^*} is the merit function of the A^* algorithm, h is the heuristic of the planner and $size$ is the size of the search tree at the moment the state is added to the priority queue. The priority of the planner functions in lexicographic order, meaning that the expanded state always has the minimum f_{A^*}, and if there are multiple states with equal f_{A^*} then the tiebreak is decided based on which state has the lower heuristic value, and if those are tied as well then it is based on the size of the search tree. This is beneficial to us, as we will be manipulating the value of the heuristic function and therefore, having the ability to prefer the states with lower heuristic values without breaking the A^* guarantee 2.2.1 is a significant detail (see *Tiebreaking* 5.3.4).

The SymbolicPlanners library itself is built on top of a library called PDDL.jl [12]. This is a Julia library which serves to parse the PDDL language and simulate planning problems. It allows loading domain and problem files, checking what actions are available in what state, executing these actions on states to transform them into new ones etc.. This is key for the SymbolicPlanners library as it provides the "low-level" functions needed to interact with and solve planning problems.

SymbolicPlanners provides numerous heuristics to aid the aforementioned planners. Importantly, these heuristics do include the deletion-relaxation heuristic $h_{\max}$ but do not include the $h_{\mathrm{LM-Cut}}$ heuristic. During the making of this work, we found out that $h_{\mathrm{LM-Cut}}$ would be pivotal for our work (*Choosing input heuristics* 5.1.1), so we chose to implement the heuristic ourselves.

4.2 Implementation

Since the SymbolicPlanners library has an implementation of $h_{\max}$ it would be an obvious choice to use the already existing version of the algorithm in the implementation of the $h_{\mathrm{LM-Cut}}$ heuristic, which works with $h_{\max}$ values. SymbolicPlanners has a function which creates a relaxed planning graph from which it computes various deletion-relaxation heuristics (such as $h_{\max}$ and h_{add}). This function crucially does not allow the modification of action costs in a simple way. As this is something necessary for $h_{\mathrm{LM-Cut}}$ to function, the first step was modifying this graph-building function. The modified variant takes a dictionary mapping actions to their costs as an argument, allowing simple outside manipulation. The algorithm itself is then built as follows. We

first extract the initial costs of actions into a dictionary c. Then we add the actions $\mathrm{o}_{\mathrm{init}}$ and $\mathrm{o}_{\mathrm{goal}}$ into this dictionary and set their value to 0. Then we use the altered graph-building function from before to get the δ function and the $h_{\max}$ value of the initial setup. If the $h_{\max}$ value is infinite, then the problem is unsolvable and the algorithm terminates. Otherwise, it enters a loop. This loop begins by finding the supporters of each action, by cycling through all of the preconditions $pre(\mathrm{o})$ of every function o and finding the one with the highest value in the δ function. Then the justification graph is created. Here we represent it as a list of edges. These edges are created by cycling through all of the actions and all of the facts in the add effect $a \in add(\mathrm{o})$ and adding all possible triplets of $(supp(o), a, o)$ to the justification graph list. We then find the N_* partition by performing a modified breadth-first search from the new goal fact $\mathcal{G}$ (as a shortcut we do not have the $\mathcal{G}$ fact at all, and simply start the search by searching the edges labelled $\mathrm{o}_{\mathrm{goal}}$). We maintain a queue from which we take a fact, then we cycle through the justification graph to find edges that contain this fact as the end point, and if they start outside of N_* and their cost is 0 then we add the start fact of the edge to N_* and enqueue it. In the same manner, we find the N_0 partition, we start by searching edges labelled $\mathrm{o}_{\mathrm{init}}$ and then perform a BFS through the graph using a queue. The condition of adding a fact to N_0 is that it is neither in N_0 nor in N_*, the cost of the edge can be non-zero. With the two partitions, we find the landmarks by again cycling the graph and finding which edges start in N_0 and end in N_*, of those we select the one with the lowest cost $c(\mathrm{o})$, call this cost σ, lower the cost of the all of the landmark actions in the c dictionary by σ and then run a new iteration by building the relaxed planning graph again with the new cost dictionary. This stops when the $h_{\max}$ cost returned by the relaxed planning graph function is 0 and the output is the sum of all σ throughout all iterations.

4.3 Optimization

The above-mentioned implementation works and produces correct values of the $h_{\mathrm{LM-Cut}}$ heuristic. It was made to mimic the definition of the heuristic closely, in order to be correct, but this has made it very slow. The computational speeds of heuristics are key as they allow the planner to expand more nodes, and are especially key for our approach which combines more heuristics. In consequence, we would like to discuss the optimizations and speedups we achieved.

The main problem is the justification graph. Having to loop through the entirety of it to find an edge which matches a fact is extremely inefficient. At the same time the graph needs to be traversed both forwards (during the finding of N_0) and backwards (during the finding of N_*) and it is difficult to achieve that in a single data structure without resorting to some level of inefficient searching. The solution we used was to build two graphs, one for forward searching, and one for backwards searching.

The graph for backwards searching is simpler, it consists of an array of arrays of integers which is exactly as long as the number of all facts. At the index of each fact, there is stored an array of indexes. These indexes detail which facts are the start points of edges which contain that given fact. This means during backwards search we can simply index into the array and get a list of indexes to which we can move. During the creation, we also right away check the costs of the actions which connect these facts, and we only allow insertion into the backwards justification graph when the cost of the action is 0. This means we do not have to constantly recheck the prices of actions during the backwards search. The forwards justification graph is an array of arrays of tuples which is also as long as the number of facts. At the index of each fact, there are stored tuples that detail to what actions the given fact is a supporter, and what facts those supported actions lead to. The tuple has the form $(\mathrm{o}, [\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_\mathrm{n}])$ where o is the label of the supported action, and $[\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_\mathrm{n}]$ are the end point facts of that supported action. During forwards search this allows us to simply index any given fact into this graph to find out where we can continue the search. Those were the major changes to the algorithm which were also the cause of the most significant speedup. Other optimizations were also made. The finding of landmarks was moved from a standalone operation to the finding of the N_0 partition. During the finding of it, we are sure that the fact we are currently searching is in N_0 and we are looking at what facts are connected to it, therefore, if we find that the connected fact is in the N_* partition, we can add it to the set of landmarks right away. The creation of the forward and backward justification graphs was altered so that it happened with just one cycle through all of the actions, reducing the necessary work and improving cache hits. The finding of supporters was changed to use an inbuilt Julia argmax function instead of a simple cycle through the preconditions.

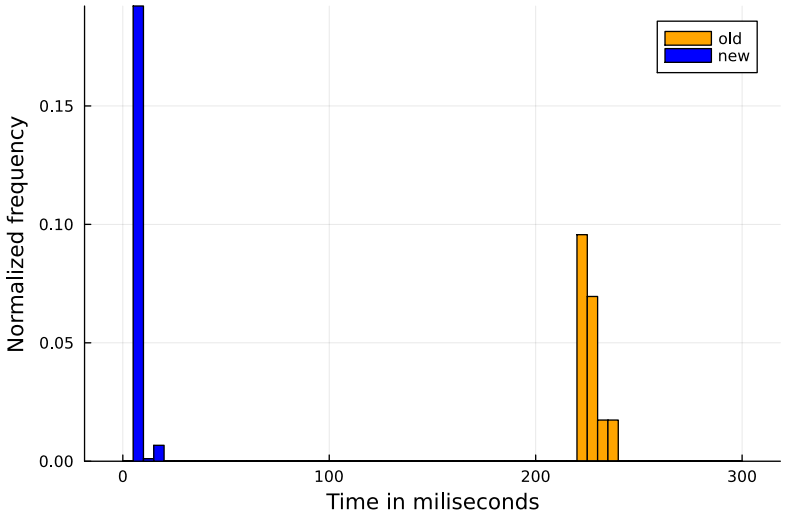

Figure 4.1: Comparison of implementations of $h_{\mathrm{LM-Cut}}$

In figure 4.1 are the histograms comparing the benchmarked speeds of the new versus the old implementation on one larger blocksworld problem. The new implementation performed significantly better. The median was lowered from 225.23 milliseconds to just 6.26 milliseconds. It should also be noted that the frequencies in the figure are normalized. This is because the old implementation was only able to compute the heuristic 23 times, compared to 743 times for the new implementation.

Chapter 5

Experimental evaluation

After constructing the admissible heuristic h_{ANH} as described in the previous chapter, we move to test and evaluate its performance.

5.1 Preparation

5.1.1 Choosing input heuristics

Choosing the input heuristics is an important part of the h_{ANH} heuristic, its performance can change significantly based on their properties. In the testing so far we have limited ourselves to two heuristics, this was done to keep the entire process simpler and to see which heuristics would work well with h_{ANH}. The only necessary constraint we have on the heuristics is that they must be admissible so that the A^* guarantee 2.2.1 is satisfied. What we want from these heuristics is primarily giving "good information". This means representing as accurately as possible which states are closer to the goal and which are not. Secondarily we would prefer the heuristics evaluate on any given state in a reasonable time, as we will be evaluating multiple on each state.

The first input heuristic is always the zero heuristic h_0. While this heuristic carries no information of its own as it can not differentiate between states, it is admissible and evaluated instantly. Since the h_{ANH} heuristic outputs a convex combination of the input heuristics, the h_0 heuristic is used as a way the reduce the value of the other heuristics if necessary. Without it the lowest possible value for h_{ANH} to output would be the minimum of the non-zero heuristics, restricting the value range.

The second chosen heuristic was the $h_{\text{LM-Cut}}$ heuristic. $h_{\text{LM-Cut}}$ is admissible and provides good information by approximating the true cost to goal well. This is important as $h_{\text{LM-Cut}}$ is going to be providing the upper bound of the output of the h_{ANH} heuristic. The computation times of the $h_{\text{LM-Cut}}$ heuristic proved not to be a problem, in large part due to the optimizations we have made to our implementation 4.3.

5.1.2 Evaluation environment

The primary domain for training was the blocksworld domain. Blocksworld had a lower percentage of states that were indistinguishable in the representation of the graph neural network. h_{ANH} struggles greatly with these states, as it needs to identify the states that are on the optimal path.

The graph neural network model representing the h_{ANH} heuristic is generalised to work with any underlying representation that NeuroPlanner supports, but all testing was done with the HyperGraph [6] representation (as described before). This was chosen as it was faster than the alternatives and it also performed well during training.

The model itself was created by the through the Mill.jl [9] library. This library can take on the input of an initial state from a representation and makes a model that can accept (work with/process) the passed state. With the model created this way, we replace the final layer with our own custom one.

We chose the L^* loss function to use in training.

5.2 Verifying the concept

To ensure that the concept of the h_{ANH} heuristic works correctly and also to show closely how the heuristic function works, we will now go through an example where the training set of the model is a single small blocksworld problem. The figures below showcase a search tree of an A^* planner with the $h_{\text{LM-Cut}}$ heuristic. The nodes in red are the nodes of the optimal path (the trajectory), and the grey nodes are the non-optimal nodes. The search is here visualised top to bottom, meaning the very top red state is the start state, and the one at the bottom is the goal state. The arrows denote which states were expanded from which. Finally, the numbers in the nodes are the heuristic values of those nodes, in this case, $h_{\text{LM-Cut}}$ values.

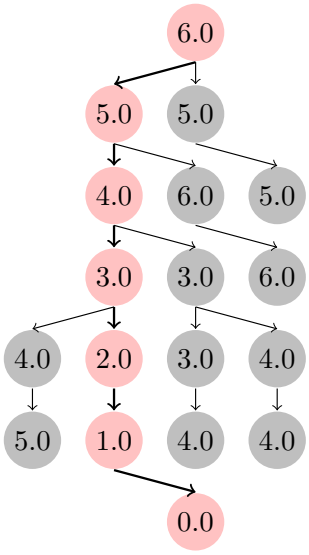

Figure 5.1: A search tree of an A^* planner with $h_{\text{LM-Cut}}$

The h_{ANH} heuristic outputs from its model convex coefficients for its input heuristics, which then get combined to create the final heuristic value. In this case, the model outputs coefficients α_1 and α_2 which get multiplied with $h_{\text{LM-Cut}}$ and h_0 respectively. The output is then:

$$h_{ANH} = \alpha_1 * h_{\text{LM-Cut}} + \alpha_2 * h_0 \tag{5.1}$$

But since the h_0 heuristic is always 0, we can simplify to:

$$h_{ANH} = \alpha_1 * h_{\text{LM-Cut}} \tag{5.2}$$

And as such we can also look at finding the h_{ANH} heuristic for a given state as finding the α_1 coefficient for $h_{\text{LM-Cut}}$. The h_{ANH} heuristic takes as its minibatches $h_{\text{LM-Cut}}$ search trees, so we can demonstrate how it works on the search tree found 5.1.

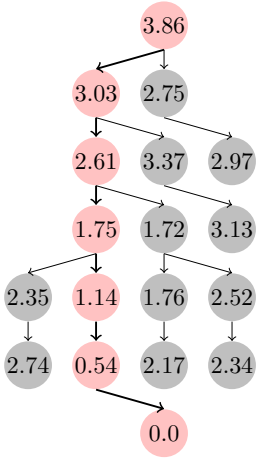

Figure 5.2: h_{ANH} output before training

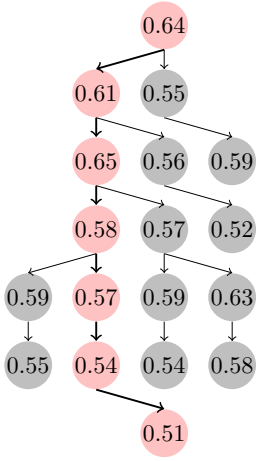

Figure 5.3: h_{ANH} α_1 coefficient before training

When a new model is created it is initialized with random weights. On the left 5.2 are the heuristic values of h_{ANH} with these newly created weights. On the right 5.3 are the values of the α_1 coefficient for every state before training, signifying the percentage of the value of $h_{\text{LM-Cut}}$ in that state.

The L^* loss function that is used during training penalizes unfulfilled inequalities. Here, the inequalities are that the states on the optimal path (red) should have a lower heuristic value than the non-optimal states (grey). If we look at the results after training, we can see that this is achieved.

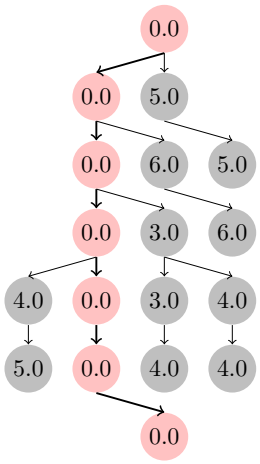

Figure 5.4: h_{ANH} output after training

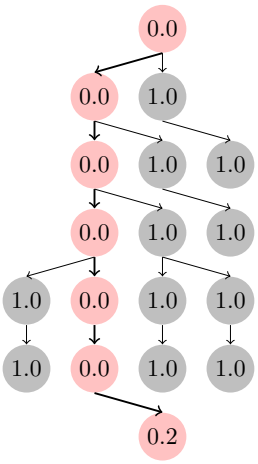

Figure 5.5: h_{ANH} α_1 coefficient after training

The h_{ANH} heuristic both successfully reduces the heuristic value of the optimal states to zero, it also raises the heuristic values to the maximum it can (the value of the $h_{\text{LM-Cut}}$ heuristic) 5.4. On the right 5.5 we can see that this is achieved by reducing the α_1 coefficient to 0 for optimal states, and 1 for non-optimal. Note that even though the α_1 coefficient for the goal state is not 0, this does not affect the heuristic output as $h_{\text{LM-Cut}}$, like all admissible heuristics, has a value of 0 in the goal state. Finally, we can run the A^* planner again, this time with the trained h_{ANH} heuristic:

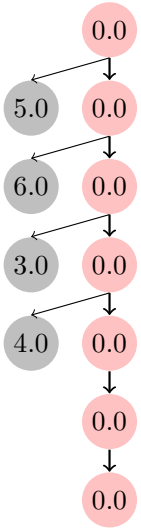

Figure 5.6: A^* planner with trained h_{ANH} heuristic

The trained h_{ANH} heuristic did not expand any non-optimal nodes, improving on the $h_{\text{LM-Cut}}$ search.

5.3 Dead ends

In the making of this work, many dead ends to progress were encountered. They are listed here both so that future works may avoid them and illustrate how this work came to be.

5.3.1 $h_{\max}$

First choice to compliment h_0 was the $h_{\max}$ heuristic, a deletion-relaxation heuristic. It was chosen because it is simple and admissible. This choice proved problematic as it turned out that $h_{\max}$ did not provide a wide enough range of values. Especially on small instances, $h_{\max}$ assigned most states one of two values, either $h_{max}(s_{init})$ or $h_{max}(s_{goal})$. The number of states for which $h_{\max}$ provided a value different from these were a small minority. This led to the model often failing to learn at all, because the information provided by $h_{\max}$ was simply not enough. For example, here are histograms of heuristic values provided by $h_{\max}$ and $h_{\mathrm{LM-Cut}}$ during searches of a larger blocks problem:

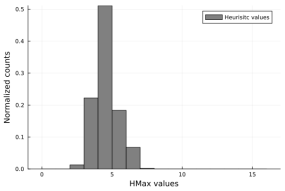

Figure 5.7: Histogram of $h_{\max}$ values

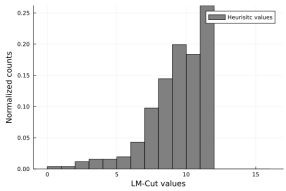

Figure 5.8: Histogram of $h_{\mathrm{LM-Cut}}$ values

$h_{\mathrm{LM-Cut}}$ provided a wide range of values, focused mostly on higher values 5.8. $h_{\max}$ provided a narrow range of heuristic values concentrated mostly into one of three numbers 5.7. This made it difficult to train the h_{ANH} model, as the heuristic output of h_{ANH} was bounded at that time by the maximum $h_{\max}$ value in that state. This made the difference between states on the optimal path and off the optimal path quite small and caused the heuristic to perform poorly. Also, note that the counts in the histograms are normalized. As $h_{\max}$ is faster to evaluate and gives less information, it expanded an order of magnitude more states, yet still gave less varied outputs

5.3.2 Domain choice

The initial primary domain for testing was not blocksworld, but ferry. During training with ferry, however, we were not able to achieve consistent results, even when training on a single problem. The goal was to get the results described above 5.2 - heuristic values of 0 on the trajectory and heuristic values equal to $h_{\mathrm{LM-Cut}}$ elsewhere, as that should always be the result of the $L*$ 3.3.2 loss function on the minibatch. However, commonly there would be optimal states that had non-zero heuristic values and non-optimal states with heuristic values of zero. Upon further investigation it turned out that the graph neural network was not able to properly distinguish between several of the states in the minibatch - it assigned them the same representation. During training then this caused problems as one representation could be both on the optimal path and off of it, causing these irregularities. The expressiveness of the underlying model is not within the scope of this work and is currently being addressed in other works [8]. This has been solved by using the blocksworld domain instead of ferry, as ferry had a large problem with these indistinguishable states, which is significantly reduced in blocksworld.

5.3.3 Minibatches

The $L*$ 3.3.2 loss function is normally assigned a specific minibatching method by the Neuro-Planner.jl library, This method usually takes the trajectory and expands every state on it once.

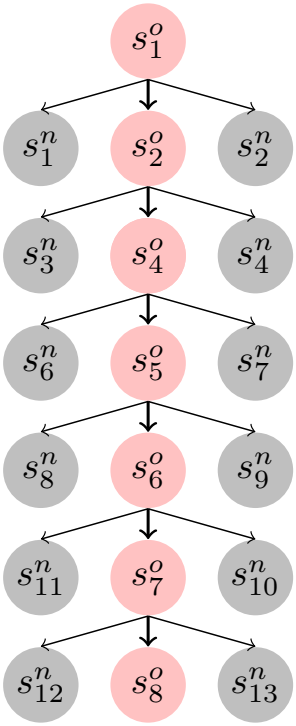

Figure 5.9: And example of a $L*$ minibatch

The inequalities of $L*$ are then set so that the states on the optimal path have a lower heuristic value than the non-optimal states that have been encountered during the search so far. For example optimal state s_{opt2} in the figure has a lower heuristic value than states s_{1-2} and optimal state s_{opt6} has lower heuristic values than states s_{1-9}. This approach has several benefits. The time to create these minibatches is relatively short and in theory these inequalities should be enough to force the A^* planner to only expand states on the optimal path, as their heuristic values would be only compared to these adjacent states. This approach works well when creating non-admissible heuristics focused on coverage (that is how NeuroPlanner operates by default). However, the A^* planner also takes into account the distance from the start state. Through certain combinations of low values of $h_{\mathrm{LM-Cut}}$ in non-optimal states and problems which have long optimal paths, non-optimal states can be expanded. This causes significant problems as these new states gained from this expansion are outside of the training set, and the h_{ANH} heuristic does not know what values to assign these states. If it assigns low values, then these states can also be expanded, compounding the problem and lowering performance.

The solution to this is that the minibatch is modified. During the creation, the problem in question is solved with the A^* planner with the $h_{\mathrm{LM-Cut}}$ heuristic. The entire search tree produced from this process is then used as a minibatch for training. This in the majority of cases increases the number of non-optimal states in the minibatch and thus the number of inequalities generated. This gives some leeway if a non-optimal state is expanded, as there is a chance that the new state is also in the training set. It also increases the generalization capabilities of the heuristic, assuring that results improve even when a state that has never been seen before is encountered.

5.3.4 Tiebreaking

At the end of testing, after all of the above issues were addressed, the h_{ANH} was still consistently performing worse than $h_{\mathrm{LM-Cut}}$ in term of expanded nodes when training and testing on a single planning problem. After closer inspection, this turned out to be caused by the model outputting real numbers instead of integers. Since most heuristics are approximating cost to goal which is the number of steps to reach the goal, they are only outputting whole numbers. Since h_{ANH} is represented by a graph neural network model, which is evaluated by doing a large number of calculations with floating point numbers, its output too is a floating point number. This causes problems as most states on the trajectory are going to have a heuristic value of zero and thus their

merit function will be equal to the integer distance from the start.

$$f_{A^*} = g + h; \tag{5.3}$$

$$h = 0 \tag{5.4}$$

$$f_{A^*} = g \tag{5.5}$$

Whereas the non-optimal states will have their merit function composed of an integer distance from the start, but also a heuristic function that is approaching the value of the $h_{\mathrm{LM-Cut}}$ from below, but not being equal to it. A state with $h_{\mathrm{LM-Cut}}$ value of 6 could have a h_{ANH} values of 5.999999998. This then means that in cases where a tie should occur, it does not. An optimal state with merit function $f_{A^*}(s) = 12 + 0$ would not tie a non-optimal state with a merit function value of $f_{A^*}(s) = 6 + 5.999999998$, even though we would like it to. This is because during tiebreaking, we have a clear way how to break the tie in favour of the optimal state, that being to always favour the state with the lower heuristic value. This was solved by simply rounding all of the results of the h_{ANH} model after training was done, thus forcing the tiebreaks and also making the heuristic behave more like a typical heuristic (by giving discrete outputs). This boosted performance significantly and led to h_{ANH} outperforming $h_{\mathrm{LM-Cut}}$ in the number of expanded nodes on the training set.

5.4 Performance

We evaluated several domains with multiple different settings of the h_{ANH} neural network model. We will be showing the performance of the best-performing model and comparing it to the $h_{\mathrm{LM-Cut}}$ heuristic on the same problems from the same domains. $h_{\mathrm{LM-Cut}}$ is the most sensible comparison as h_{ANH} is essentially expanding the heuristic with the addition of h_0.

domain	**coverage** h_{ANH}	**coverage** $h_{\mathrm{LM-Cut}}$
blocks	0.68	0.72
elevators_00	0.64	0.80
ferry	0.68	0.68

Table 5.1: Comparison of coverage

The table 5.1 above shows the coverage of each heuristic across the tested domains, i.e. what percentage of the planning problems in that domain the A^* planner could solve with the given heuristic. This data only covers problems that were not part of the training set for the h_{ANH} neural network model. Across the board, the $h_{\mathrm{LM-Cut}}$ heuristic was able to solve more problems than the h_{ANH} heuristic. Moreover, the h_{ANH} heuristic was not able to solve any problems that the $h_{\mathrm{LM-Cut}}$ heuristic did not solve.

The performances of each heuristic also varied in the problems that were solved by both:

domain	**total expanded** h_{ANH}	**total expanded** $h_{\mathrm{LM-Cut}}$
blocks	34948	30537
elevators_00	1064	959
ferry	4884	4068

Table 5.2: Number of expanded states comparison

The table 5.2 showcases the total number of states expanded by the given heuristic across all problems solved by both heuristics in that domain. Even when both heuristics solved the problem the h_{ANH} heuristic expanded more states.

Furthermore, even when the number of expanded states was similar (such as in the elevators_00 domain), the solution times were vastly different 5.3. The h_{ANH} heuristic is slower across the board, having to both calculate the neural network representation and the value of the $h_{\mathrm{LM-Cut}}$ heuristic for any given state.

domain	solution time h_{ANH}	solution time $h_{\text{LM-Cut}}$
blocks	658.45	275.80
elevators_00	185.50	54.59
ferry	45.96	11.06

Table 5.3: Solution times comparison [s]

5.5 Discussion

The coverage gap can be better understood by looking at the problems that $h_{\text{LM-Cut}}$ managed to solve, while h_{ANH} did not, specifically at the total number of expanded states:

domain	total expanded h_{ANH}	total expanded $h_{\text{LM-Cut}}$
blocks	20717	32995
elevators_00	680	1085
ferry	-	-

Table 5.4: Expanded states on problems only $h_{\text{LM-Cut}}$ solved

Here 5.4 and in the table of times 5.3 the main advantage $h_{\text{LM-Cut}}$ has over h_{ANH} is visible. $h_{\text{LM-Cut}}$ is significantly faster than h_{ANH} and thus can expand more states in the same amount of time, improving the odds of finding the goal state. On the problems where h_{ANH} did not find the goal, but $h_{\text{LM-Cut}}$ did, $h_{\text{LM-Cut}}$ expanded more state even though it stopped expanding after finding the goal state, while h_{ANH} kept expanding until it reached the limit time of the A^* planner. This may be confusing as h_{ANH} expanded more states than $h_{\text{LM-Cut}}$ on problems they both solved 5.2. This is because the problems that were solved by both heuristics are in some sense "easier" than the ones that only $h_{\text{LM-Cut}}$ solved. Note that in the ferry domain, there is no data as both heuristics had the same coverage.

The speed difference between the two heuristics is fundamentally unsolvable with our current approach, as h_{ANH} must also evaluate $h_{\text{LM-Cut}}$ as part of its process. Therefore the h_{ANH} heuristic has to focus on providing better information for the search. As can be seen on 5.2, this is not the case. The A^* planner with h_{ANH} has to expand more states to reach the goal state than A^* with $h_{\text{LM-Cut}}$. This can be due to a variety of reasons. The problem of indistinguishable problems we discussed as being between multiple states of a problem can also happen across multiple problems. If two states across two different states have the same representation in the neural network model of h_{ANH}, and one is on the optimal path and one is not, then it will cause problems. Also likely is the cause being poor generalization of the model. The model has to learn patterns that signify states on the optimal path or "good" states in general. If it does not learn these patterns then performance can be poor. This is also not necessarily a problem of new unseen states performing sub-optimally.

domain	total expanded h_{ANH}	total expanded $h_{\text{LM-Cut}}$
blocks	28692	23799
elevators_00	1323	939
ferry	8943	7378

Table 5.5: Results of heuristics on problems used in the training of the model

The results on states the h_{ANH} heuristic had as part of its training set are not too different from those where the states were new 5.2. This is unusual and since we have shown that the approach can work on singular problems, suggests that the h_{ANH} approach has trouble learning appropriate weights for a set of problems even with perfect information.

This means that the h_{ANH} heuristic in its current state is slower than the $h_{\text{LM-Cut}}$ heuristic and gives worse information. There, however, exist proposals with which further work could improve the h_{ANH} heuristic. First adding more and better input heuristics. The h_{ANH} heuristic functions upon

a framework which allows for easy addition of any number of admissible heuristics. In this work, we were limited by what implementations existed of the admissible heuristics. Here $h_{\mathrm{LM-Cut}}$ and h_0 represent a sort of worst-case scenario ($h_{\max}$ notwithstanding), as we only have one heuristic providing information. With more input heuristics, the model would be capable of being more expressive as it could choose from more values of heuristics, and presumable have an increased upper bound as well. Ideally, these new heuristics would be of a different family/approach than $h_{\mathrm{LM-Cut}}$ or $h_{\max}$, then they could give good information in states where these heuristics do not. One such candidate are pattern database heuristics [13], which compute a database of sub-problem solutions to get a heuristic value, and critically, are incomparable to landmark heuristics [14] (such as $h_{\mathrm{LM-Cut}}$), meaning we can not safely say that one dominates (always gives higher values than) the other. This is a positive, as this means that they provide different enough values for their combination to be interesting. This would increase the evaluation time of h_{ANH} even further, however, and as such another proposal is to move away from the convex combination approach. The new method would use the neural network model to determine a single input heuristics to be evaluated. The model would essentially function as a heuristic that indicates which heuristic function to evaluate. This would avoid the problem of computing multiple heuristics for a single state and sometimes allow us to skip evaluating computationally expensive heuristics altogether (when the h_0 heuristic is chosen).

Chapter 6

Conclusion

The state spaces of planning problems can be too large to be solved without heuristic search. Designing these heuristics to aid the search is difficult. Recent approaches have had success with using neural networks to find these heuristics, however, this leads to non-admissible heuristics. In this work, we introduce the admissible heuristic h_{ANH}. The admissible heuristic h_{ANH} is a heuristic function represented by a neural network model. It uses the output of the neural network model to find a convex combination of several admissible input heuristics. It is guaranteed to be admissible due to it being a convex combination of admissible heuristics and it can take advantage of the strengths of the individual input heuristics. The h_{ANH} heuristic in its current iteration consists of two heuristics, the zero heuristic h_0 and the landmark heuristic $h_{\mathrm{LM-Cut}}$. As it was necessary for the creation of the h_{ANH} heuristic, the implementation of the $h_{\mathrm{LM-Cut}}$ heuristic is also part of this work, the heuristic is described, implemented and optimized. The h_{ANH} heuristic is designed to give the states on the optimal path a low heuristic value and the states not on the optimal path a high heuristic value to expand as few as possible states during search with the A^* planner. However, in its current iteration, the heuristic is slower and expands more states than when the A^* planner is used with the $h_{\mathrm{LM-Cut}}$ heuristic. Further work is proposed on how to fix these problems. First by adding more, stronger heuristics. The h_{ANH} allows for easy addition of input heuristics into itself, and as such it can take advantage of heuristics of vastly differing approaches and strengths. Second, by altering the heuristic to always select an admissible heuristic to evaluate for the given state, reducing the computation time and expanding more states in less time.

Bibliography

1. PEVNY, Tomas. *NeuroPlanner.jl* [https://github.com/pevnak/NeuroPlanner.jl]. GitHub, 2023.
2. LIPOVETZKY, Nir. *Structure and Inference in Classical Planning*. Morrisville, NC: Lulu.com, 2014.
3. BONET, Blai; GEFFNER, Héctor. Planning as heuristic search. *Artificial Intelligence*. 2001, roč. 129, č. 1, pp. 5–33. ISSN 0004-3702. Available from DOI: https://doi.org/10.1016/S0004-3702(01)00108-4.
4. HELMERT, Malte; DOMSHLAK, Carmel. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? *Proceedings of the International Conference on Automated Planning and Scheduling*. 2009, roč. 19, č. 1, pp. 162–169. Available from DOI: 10.1609/icaps.v19i1.13370.
5. TOYER, Sam; THIÉBAUX, Sylvie; TREVIZAN, Felipe; XIE, Lexing. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research*. 2020, roč. 68, pp. 1–68. ISSN 1076-9757. Available from DOI: 10.1613/jair.1.11633.
6. SHEN, William; TREVIZAN, Felipe; THIÉBAUX, Sylvie. *Learning Domain-Independent Planning Heuristics with Hypergraph Networks*. 2019. Available from arXiv: 1911.13101 [cs.AI].
7. CHEN, Dillon Ze; THIEBAUX, Sylvie; TREVIZAN, Felipe. GOOSE: Learning Domain-Independent Heuristics. In: *NeurIPS 2023 Workshop on Generalization in Planning*. 2023. Available also from: https://openreview.net/forum?id=cTtMNEn2Kr.
8. HORCIK, Rostislav; ŠÍR, Gustav. Expressiveness of Graph Neural Networks in Planning Domains. In: *34th International Conference on Automated Planning and Scheduling*. 2024. Available also from: https://openreview.net/forum?id=pKEkSAPSGJ.
9. MANDLIK, Simon; RACINSKY, Matej; LISY, Viliam; PEVNY, Tomas. *Mill.jl and JsonGrinder.jl: automated differentiable feature extraction for learning from raw JSON data*. 2021. Available from arXiv: 2105.09107 [stat.ML].
10. CHRESTIEN, Leah; PEVNÝ, Tomás; EDELKAMP, Stefan; KOMENDA, Antonín. *Optimize Planning Heuristics to Rank, not to Estimate Cost-to-Goal*. 2023. Available from arXiv: 2310.19463 [cs.AI].
11. ZHI-XUAN, Tan. *SymbolicPlanners.jl*. 2023. Ver. 0.1.10. Available also from: https://github.com/JuliaPlanners/SymbolicPlanners.jl.
12. ZHI-XUAN, Tan. *PDDL.jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning*. 2022. PhD thesis.
13. CULBERSON, Joseph C.; SCHAEFFER, Jonathan. Pattern Databases. *Computational Intelligence*. 1998, roč. 14, č. 3, pp. 318–334. Available from DOI: https://doi.org/10.1111/0824-7935.00065.
14. HELMERT, Malte. LM-Cut : Optimal Planning with the Landmark-Cut Heuristic. In: 2009. Available also from: https://api.semanticscholar.org/CorpusID:5788925.