SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

NATIONAL PHD PROGRAM IN ARTIFICIAL INTELLIGENCE

SSD: ING-INF/05

CYCLE XXXVIII

Learning Graph Neural Network Heuristics for Numeric Planning Problems

Candidate:
Valerio Borelli
Student ID number: 2080887

Supervisor: Prof. Alfonso Emilio Gerevini
Co-Supervisor: Prof. Enrico Scala

Abstract

Numeric planning extends classical planning by allowing the representation of continuous quantities and resources, making it suitable for modeling complex real-world problems. However, the infinite state space and the complexity of numeric conditions make the derivation of effective domain-independent heuristics particularly challenging. While Graph Neural Networks (GNNs) have shown great promise in learning heuristics for classical planning by exploiting the relational structure of the problems, their application to numeric planning is still in its infancy. Existing approaches often struggle to capture the rich structural dependencies defined by numeric constraints, or they rely on architectures that fail to generalize effectively across different problem sizes.

In this thesis, we propose a novel GNN-based architecture designed to learn heuristic functions for numeric planning problems. Unlike previous works that treat numeric fluents as isolated node features, our approach explicitly encodes grounded numeric conditions (both preconditions and goals) as edges in the graph, integrating numeric values directly into the message-passing mechanism. This allows the network to reason about the logical and arithmetic relationships between objects, leading to more informative distance estimates.

To support the integration of deep learning with automated planning, we introduce LeapNP (Learning and Planning Framework for Numeric Problems),

a modular and extensible Python framework based on Unified Planning. Furthermore, we address the computational bottleneck of evaluating neural networks during search by proposing novel search algorithms tailored for GPU acceleration. Specifically, we introduce Multiple Evaluation Best-First Search (MBFS), which exploits batch processing to evaluate states in parallel, and Adaptive Width Best-First Search (AWBFS), a dynamic algorithm that adjusts the search width based on heuristic feedback to escape plateaus efficiently.

Experimental results on domains from the IPC 2023 Numeric Track demonstrate that our architecture significantly outperforms both traditional heuristics and state-of-the-art learning-based approaches in domains with complex numeric structures. Moreover, we show that AWBFS effectively leverages the throughput of modern GPUs, achieving higher coverage and better plan quality compared to standard greedy search strategies.

Contents

1 Introduction **1**

1.1 Topic of the Thesis 1

1.2 Structure of the Thesis 3

2 Background **5**

2.1 Foundations of Automated Planning 5

2.1.1 Classical Planning 5

2.1.2 Numeric Planning 9

2.2 Graph Neural Networks 10

3 Related Work **13**

3.1 Graph Neural Networks for Classical Planning 13

3.1.1 GNNs as general policies 13

3.1.2 GNNs as heuristic functions 14

3.1.3 State encodings and Expressiveness 15

3.2 Learning-based approaches for Numeric Planning 17

3.2.1 Numeric ASNets 17

3.2.2 Graph Kernels for Numeric Planning 18

3.3 Frameworks for Planning and Learning 19

3.3.1 Traditional Planning Systems 19

3.3.2 Lowering the entry barrier to planning, the Unified Planning Library . . . 20
3.3.3 Planners in Python, Pyperplan and NyX . . . 21
3.4 Parallel Search Strategies . . . 22
3.4.1 Parallel Best-First Search . . . 22
3.4.2 Adaptive Beam Search . . . 23
3.4.3 Exploration and Width-based strategies . . . 24

4 Learning Numeric Planning Heuristics with Graph Neural Networks 25
4.1 Architecture . . . 25
4.1.1 Graph Construction . . . 25
4.1.2 GNN Construction . . . 27
4.1.3 Generalization . . . 32
4.1.4 State Encodings and Heuristic Computation . . . 34
4.1.5 Training Procedure . . . 36
4.2 Numeric Planning in Python: LeapNP . . . 38
4.2.1 Components . . . 38
4.2.2 Modules . . . 41
4.2.3 Search Algorithms . . . 46
4.3 Adaptive Width Best-First Search . . . 53
4.3.1 Dynamic Width Adaptation . . . 54
4.3.2 Memory Constraints with Learning-Based Heuristics . . . 55
4.3.3 Architectural Considerations . . . 56
4.3.4 Algorithm . . . 58
4.3.5 Case Study: The First-Order Farmland Domain . . . 60
4.3.6 Case Study: The Expedition Domain . . . 63

5 Experimental Analysis **67**

5.1 Benchmarks and Training Setup 68

5.2 Heuristics 72

5.2.1 Heuristics Performance and Coverage Analysis 73

5.2.2 Comparison with h^{ccWLF}_{rank} 76

5.2.3 Scaling and Generalization Analysis 78

5.3 Multiple Evaluation Algorithms Trade-offs 82

5.3.1 MBFS and DBFS 82

5.3.2 Optimization vs. Speed 85

5.3.3 Search vs. Policy 86

5.4 Empirical Analysis of AWBFS 88

5.4.1 Comparison with KBFS 90

6 Conclusion and Future Work **94**

6.1 Summary of Contributions 94

6.2 Limitations 96

6.3 Future Work 98

6.3.1 Numeric HER 98

6.3.2 Learning Numeric Macro-Actions 99

6.3.3 Multi-Queue Search Strategies 99

A The Numeric Gripper Domain **101**

Chapter 1

Introduction

1.1 Topic of the Thesis

Automated planning is a central area of Artificial Intelligence concerned with the development of algorithms capable of selecting and organizing actions to achieve specific goals. While classical planning has reached a high level of maturity, many real-world applications, such as robotics, logistics, and process control, require reasoning about continuous resources, time, and physical parameters. Numeric planning addresses these needs by extending the classical paradigm with numeric state variables. However, this expressiveness comes at a cost: the state space becomes infinite, and the problem complexity increases significantly, rendering many traditional heuristic techniques less effective or computationally expensive.

In recent years, several works have shown that learning-based approaches can be used to solve planning problems, with models that can be used effectively either as heuristic functions, as standalone planning systems, or as value functions for general policies. In the context of classical planning, interesting

results have been achieved by different approaches: PlanGPT [40], which uses a GPT-based model to find a plan given a domain and a problem; general policies approaches [45, 46, 49], which use different types of graph neural network architectures to learn a generalized policy for a classical planning domain; graph kernel approaches that mimics the expressive power of Graph Neural Networks, while significantly reducing the computational overhead [11]; and by heuristic approaches such as hypergraph networks [43] and relational heuristic networks [32].

More recently, specific attention has been directed towards learning for numeric planning. The Numeric ASNets architecture [50], building on the work by Toyer et al. [49], represents a significant step forward, enabling reasoning about numeric fluents and their interactions through specialized modules and a dynamic exploration algorithm. However, ASNets provides a neural architecture trained as a general policy to select actions directly, rather than a heuristic function to guide a systematic search. A second notable approach is the one presented by Chen et al. [11], which builds on the framework introduced by Chen et al. [10]. They propose the use of graph kernels and optimization techniques tailored for numeric planning. This method demonstrates superior efficiency and generalizability compared to traditional GNN approaches, showing competitive performance against domain-independent numeric planners.

Despite these advancements, the application of GNNs to learn *heuristic functions* for numeric planning remains relatively unexplored. Building upon the GNN approach for learning general policies proposed by Ståhlberg et al. [46], this thesis investigates the main issues that need to be addressed to exploit GNN models for solving numeric planning problems. We propose a novel GNN architecture that integrates numeric conditions directly into the topological structure of the graph. By modeling conditions as edges that carry numeric information,

our network can learn to reason about the satisfaction of constraints and the ”distance” to numeric goals more effectively than existing methods.

Furthermore, this thesis addresses the significant engineering barriers that currently limit research at the intersection of Deep Learning and Numeric Planning. State-of-the-art planners are implemented in low-level languages and highly optimized for search speed; while efficient, these systems are often rigid and ill-suited for the rapid prototyping required by learning-based methodologies. To bridge this gap, we introduce LeapNP (Learning and Planning Framework for Numeric Problems), a lightweight and fully modular framework implemented in Python. Designed to natively support numeric tasks, LeapNP lowers the technical entry barrier for researchers, enabling the seamless integration and benchmarking of deep learning components within a standard planning cycle.

Finally, to mitigate the computational bottleneck of neural inference and improve search robustness, we propose a novel search algorithm tailored for GPU acceleration. Adaptive Width Best-First Search (AWBFS) is designed to address the structural limitations of standard Greedy Best-First Search (GBFS) [6], which often gets stuck in search plateaus, where the heuristic offers no guidance. By dynamically adapting its exploration strategy, AWBFS increases the search width only when the search hits a plateau, specifically, when the heuristic function stops improving, allowing it to broaden the frontier and escape uninformative regions. Furthermore, to ensure feasibility when using memory-intensive learning-based heuristics, AWBFS incorporates a mechanism to track GPU memory usage, ensuring the search never exceeds the available hardware limits.

1.2 Structure of the Thesis

This thesis is organized as follows:

- **Chapter 2** establishes the theoretical foundations of the thesis. It provides the necessary background on Automated Planning, formally defining Classical and Numeric Planning tasks. Furthermore, it introduces Graph Neural Networks (GNNs), describing the fundamental message-passing mechanism, and discusses Heuristic Search, with a specific focus on parallel evaluation and GPU acceleration strategies.

- **Chapter 3** discusses related work, positioning the thesis within the state of the art. We analyze existing learning-based approaches for both classical and numeric planning (including Numeric ASNets and Graph Kernels), discuss the limitations of current planning frameworks, and examine search algorithms designed for learned heuristics.

- **Chapter 4** details the core contributions of this thesis. We first present our novel GNN architecture, describing how numeric states are encoded into graphs. Next, we introduce the LeapNP framework, detailing its modular components. Finally, we describe the implementation of our novel search algorithms, MBFS and AWBFS, discussing the theoretical and practical considerations behind their design.

- **Chapter 5** presents the experimental analysis. We describe the training setup and the benchmarks used. We then perform a comparative analysis of our heuristic against traditional and learning-based baselines. Finally, we evaluate the performance of the proposed search algorithms, highlighting the benefits of parallel evaluation and adaptive width search.

- **Chapter 6** concludes the thesis, summarizing the main findings and outlining directions for future research.

Chapter 2

Background

2.1 Foundations of Automated Planning

The term "automated planning", or planning for short, refers to a set of languages, algorithms, and systems whose purpose is to plan, that is, to select and organize actions in order to achieve specific objectives, called goals. Planning is part of symbolic artificial intelligence, as it is based on formal languages composed of symbols.

2.1.1 Classical Planning

Classical planning represents the most common and simplified type of planning. It assumes a simplified and idealized model of the world, which makes reasoning about plans computationally tractable, allowing the development of general-purpose planning algorithms and heuristics. In the classical planning paradigm, the environment is represented as deterministic, fully observable, static, and discrete.

- **Deterministic**: every action has a single, predictable outcome.

- **Fully observable**: the planner has complete knowledge of the world state at all times.
- **Static**: the world does not change unless the planner performs an action.
- **Discrete**: the world is described through a finite set of states and actions.

Classical planning languages, such as **STRIPS**(STanford Research Institute Problem Solver)[16] and **PDDL**(Problem Domain Definition Language)[23], standardize how domains and problems are described. These formalisms express the logical structure of actions and enable the use of generic planning algorithms.

We will now describe a classical planning problem in the **PDDL**[23] formalism. We first introduce the notion of lifted classical planning problem, and then define its grounding. A lifted classical planning problem is a pair $\Pi = \langle D, I \rangle$, where D is the domain that defines the problem, and I is a specific instance for the domain. The domain D is a tuple $\langle \mathcal{P}, \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{P}$ is a set of boolean fluents (also called predicates), $\mathcal{T}$ is a set of types, and $\mathcal{A}$ is a set of action schemas. Boolean fluents are defined in terms of their name and list of variables typed in $\mathcal{T}$. An action schema $a \in \mathcal{A}$ is a tuple $\langle Par(a), Pre(a), Eff(a) \rangle$, with $Par(a)$ being a list of parameters of the action schema (lifted variables typed in $\mathcal{T}$); $Pre(a)$ is a conjunction over $\mathcal{P}$ of propositional literals, such as $p(?x)$; $Eff(a)$ are assignments over $\mathcal{P}$, such as $(p(?x) := \top)$.

Action schemas and boolean fluents can be grounded against a set of typed objects O. Intuitively, the grounding consists of generating as many actions as there are valid substitutions for the parameters, and propagating the mapping over preconditions and effects. A grounded classical planning problem is the resulting formulation. A state for a grounded classical planning problem is a full assignment to all boolean fluents; it can be represented compactly by asserting

which boolean fluents are true, and everything not expressed is assumed false (closed world assumption).

A planning instance I is a tuple $\langle O, s_o, G \rangle$ where:

- O is a set of typed objects.
- s_o is a state, called the initial state.
- G is the goal of the problem.

The goal is a set of boolean conditions over O. By definition, each boolean fluent can be seen as a Boolean condition, i.e., a propositional statement that may hold or not in a given state.

A classical planning problem can be defined in terms of its grounded representation as a tuple $\Pi = \langle s_0, A, G, P \rangle$ where:

- P is the set of boolean variables.
- s_o is the initial state of the problem.
- A is a set of actions.
- G is the goal of the problem.

In this representation, A and P are defined structurally as in the lifted representation, with the difference that all variables have been made concrete using objects from O. For instance, literals $p(?x)$ has groundings in $p(o_1)$ and $p(o_2)$ provided that o_1 and o_2 both belong to O and share the same type of $?x$ ($p(o_1)$ and $p(o_2)$ are grounded fluents). We say that a grounded action $a \in A$ is applicable in a state s iff its preconditions $Pre(a)$ are true in s. Its execution changes s according to its assignments $Eff(a)$. A plan $\pi\langle a_1, \ldots, a_n \rangle$ is a finite sequence of grounded actions, and it is considered a solution for the planning problem iff all

grounded actions it consists of are iteratively applicable in the state just before their execution, and G is true in the last state.

Different paradigms have been introduced to tackle classical planning problems efficiently over the years. Those include the use of reachability structures known as **Planning Graphs** [5], the encoding of planning instances into **Boolean Satisfiability formulas** (SAT) [33], and, most prominently, **Heuristic State-Space Search** [6].

In the context of state-space search, the planning problem is viewed as a path-finding problem within a directed graph where nodes represent states and edges represent actions. To navigate this often immense search space efficiently, algorithms rely on **heuristics**. A heuristic function, conventionally denoted as *h(s)*, estimates the cost (or distance) required to reach a goal state from a given state s. By evaluating states based on this estimate, often combined with the cost already incurred to reach s, as in the A^*[25] algorithm, the planner can prioritize expanding the most promising states first, rather than exploring them blindly. In domain-independent planning, these heuristics are derived automatically from the domain definition, usually by analyzing a simplified version of the problem.

Historically, a major breakthrough in utilizing these techniques was achieved by the **FF** system [28]. FF introduced the use of a relaxed planning graph, ignoring the delete effects of actions, to compute highly informative heuristics, allowing it to dominate the field in the early 2000s. Subsequently, the **LPG** planner [21] demonstrated the efficacy of stochastic local search techniques operating on planning graphs, offering a highly efficient, non-systematic alternative. Later, a paradigm shift occurred with the introduction of the **Fast Downward** planning system [27]. Unlike previous planners based on pure STRIPS/PDDL binary encodings, Fast Downward translates the problem into a multi-valued representation (SAS^+) and exploits causal graphs to guide the search. Fast Downward has

since become a modular platform for planning research; essentially, the **LAMA** planner [39], implemented on top of the Fast Downward framework, set a new standard for years by combining landmark-based heuristics with cost-sensitive search algorithms and causal graphs.

2.1.2 Numeric Planning

Numeric planning extends the classical paradigm by relaxing the assumption that the world is purely discrete. While classical planning deals effectively with the causal structure of problems, it lacks the ability to reason natively about resources, continuous quantities, or physical parameters such as fuel levels, battery charge, time, or spatial coordinates. In numeric planning, the state of the world is described not only by boolean propositions, but also by a set of numeric variables that can take values from an infinite domain. Consequently, the state space becomes infinite, making the problem significantly more complex, and, in the general case, undecidable [26].

Standardized by **PDDL 2.1** [18], numeric planning introduces the concept of *numeric fluents*, enabling the modeling of systems where actions depend on numeric conditions and produce numeric changes.

Formally, a lifted numeric planning problem extends the classical definition. The domain D is a tuple $\langle \mathcal{P}, \mathcal{T}, \mathcal{A}, \mathcal{X} \rangle$, in which $\mathcal{X}$ is the set of numeric fluents. Additionally, action preconditions can also contain numeric conditions, such as $\xi \geq 0$, where ξ is a numeric expression over $\mathcal{X}$, and action effects can also contain numeric effects, such as $(f(?x) := \xi)$ with ξ being a numeric expression over $\mathcal{X}$.

Analogous to the classical case, the problem can be defined in its grounded representation. A grounded numeric planning problem is a tuple $\Pi = \langle s_o, A, G, P, X \rangle$ with X being the set of numeric variables. The goal G is composed of both boolean and numeric conditions. A state for a grounded numeric planning prob-

lem is a full assignment to all grounded boolean and numeric fluents. It can be represented compactly by asserting which boolean fluent is true, and which value each numeric fluent is taking. Everything not expressed is assumed false for boolean fluents, as before, while for numeric fluents is assumed undefined.

Over the years, approaches to numeric planning have evolved to handle the conjunction between propositional logic and the arithmetic reasoning required to handle numeric planning problems. The seminal work in this field is **Metric-FF** [29], which extended the classical delete-relaxation heuristic to handle numeric variables, allowing the extraction of heuristics that estimate the numeric "distance" to the goal. The **LPG** planner also natively supports numeric quantities through its local search mechanisms [22]. More recently, the **Numeric Fast Downward** system [2] extended Fast Downward to include numeric variables in the SAS^+ formalism. The current state of the art for numeric planning is the **ENHSP** (Expressive Numeric Heuristic Search Planner)[42] planning system. ENHSP, similar to Fast Downward for classical planning, has become a modular platform for planning research. Some of its most important contributions are an interval-based relaxation technique [41] and a subgoaling relaxation tailored to identify and reason about the specific subgoals that come from numeric preconditions and goals in a numeric problem [42].

2.2 Graph Neural Networks

A GNN is a type of neural network designed to work with graph-structured data. Traditional neural networks are designed to represent data structured in grids or sequences, but they struggle in real-world datasets structured as graphs, like social networks or molecular structures [52]. In the following, we provide a quick introduction over GNN, borrowing the notation of Hamilton [24]. A GNN takes

as input a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of node features $X \in \mathbb{R}^{k \times |\mathcal{V}|}$, where k represents the number of features for a node, and generates node embeddings $z_u, \forall u \in \mathcal{V}, z_u \in \mathbb{R}^k$. These embeddings can then be used to predict properties of nodes, edges, or the entire graph. The nodes in the GNN exchange information during a message-passing iteration, with an iteration consisting of three operations:

- *Message*, in which messages are sent from each node towards its neighbors.
- *Aggregation*, in which all messages sent to a node are aggregated together in a unique message.
- *Update*, in which the node representation is updated using the aggregated message.

Each node u in the network has a hidden embedding $h_u^{(l)}$, where l denotes the current iteration (or layer) of the GNN. This embedding is updated by aggregating information from the neighborhood of u, denoted by $\mathcal{N}(u)$. The update rule can be expressed as:

$$m_{\mathcal{N}(u)}^{(l)} = \text{AGGREGATE}^{(l)} \left(\left\{ h_v^{(l)} \mid v \in \mathcal{N}(u) \right\} \right) \tag{2.1}$$

$$h_u^{(l+1)} = \text{UPDATE}^{(l)} \left(h_u^{(l)}, m_{\mathcal{N}(u)}^{(l)} \right) \tag{2.2}$$

where AGGREGATE and UPDATE are arbitrary differentiable functions. At each iteration l, the AGGREGATE function computes a message $m_{\mathcal{N}(u)}^{(l)}$ based on the embeddings of the neighbors of u, which flows from $\mathcal{N}(u)$ towards u [24]. The UPDATE function then combines this message with the previous embedding $h_u^{(l)}$ to produce the new representation $h_u^{(l+1)}$. After L iterations of the GNN, the final node embeddings are given by: $z_u = h_u^{(L)}, \forall u \in \mathcal{V}$. These embeddings can be

used for different downstream tasks, such as node classification, graph classification, or relation (link) prediction. In node classification, the goal is to assign a class label to each node based on its final embedding. In graph classification (or regression), the embeddings of all nodes are aggregated via a READOUT function to obtain a representation of the whole graph. This is then used for classification or regression purposes. In our case, the GNN performs graph-level regression, where the input is a graph encoding of the planning state and the output is a heuristic value estimating its distance to the goal. This is computed using a READOUT function over the final node embeddings. The specific aggregation and prediction mechanism is detailed in the architecture section.

Chapter 3

Related Work

3.1 Graph Neural Networks for Classical Planning

3.1.1 GNNs as general policies

In the work by Ståhlberg et al. [46, 45], GNNs are used to construct general policies, with a network that represents, instead of a graph, a relational structure defined by a graph. This is done to reflect the fact that a state s in our planning problems represents a relational structure that occurs among the objects. Such relational structure for s is defined by the set of objects $o \in O$, the set of domain predicates $p \in \mathcal{P}$, where each predicate p is a relational symbol r of the relational structure, and the atoms $q = r(o_1, ..., o_m)$ that are true in s. The set of domain predicates is fixed for a specific domain, the set of objects may change from one instance to another, and the value of the atoms is specific for the state s. These considerations lead the authors to define a graph structure where objects represent the nodes, atoms represent labeled edges, connecting objects to atoms if the object appears in the atom, and the types of predicates define the labels (or types) of these edges. For example, if $r(o_1, o_2)$ is true in a state s, the input graph

for s contains a labeled edge for $r(o_1, o_2)$ connected to both o_1 and o_2, labeled with r. Each predicate is associated with a doppelganger predicate, called "goal predicate", that is used whenever the predicate is actually used as a goal in an instance. For this reason, together with the types of arcs defined by the predicates, the relational structure also presents an arc for each such goal predicate.

For each relation r, there will be a different MultiLayer Perceptron (MLP_r). The AGGREGATE and UPDATE functions are applied to all the nodes in the GNN (i.e. to all the objects in the problem) in every layer. The READOUT function is then used to compute a value, which is used to map a given planning state *s* into a real value *V(s)*. The architecture uses one MLP_r for each relational symbol r, one MLP_U for the UPDATE function, and two nets MLP_1 and MLP_2 for the READOUT function (this will be better explained later in Algorithm 1). The key idea of this approach is to train a model on some instances, and use the trained parameters for other instances, too. The transfer of knowledge happens through the MLPs associated to each relational symbol. That is, when approaching a new instance, the architecture will use the learned parameters off-the-shelf.

3.1.2 GNNs as heuristic functions

While general policies aim to map states directly to optimal actions, another line of research focuses on learning heuristic functions $h(s)$, which estimate the cost-to-go from a given state to the goal. These learned heuristics are then integrated into standard search algorithms like Greedy Best-First Search (GBFS) to guide the exploration. This approach combines the guidance of deep learning with the systematicity of classical search algorithms.

In this area, one of the most important approaches is the GOOSE planner (Graphs Optimised fOr Search Evaluation) [10]. Addressing the limitations of previous architectures like STRIPS-HGN [43], which relied on large grounded

hypergraphs often leading to scalability issues, GOOSE proposes novel graph representations that balance expressiveness and computational efficiency. Specifically, GOOSE intorduces a subgoal graph representation, which explicitly encodes the causal dependencies between atoms and the goal, allowing the GNN to learn the cost of achieving subgoals independently. Furthermore, they propose a Lifted Graph representation. Unlike traditional approaches that instantiate every possible proposition as a node (leading to massive graphs in large problems), the lifted representation encodes the domain schema (predicates and action types) and the objects separately. This allows the network to process instances of arbitrary size without the memory explosion associated with full grounding.

The GNN backbone of GOOSE aggregates information across these graphs to predict a scalar heuristic value. Empirical results show that GOOSE significantly outperforms previous state-of-the-art learned heuristics in terms of generalization to larger problems, demonstrating that carefully engineered graph encodings are crucial for scaling GNNs to complex planning tasks.

3.1.3 State encodings and Expressiveness

A critical aspect of applying GNNs to planning is the expressiveness of the graph encoding, i.e., the ability of the network to distinguish between non-isomorphic states that have different distances to the goal. As discussed in the theoretical analysis by Ståhlberg et al. [46] and related works by Abboud et al. [1], standard Message Passing GNNs are limited in their discriminative power. Specifically, they are as powerful as the 1-Weisfeiler-Lehman (1-WL) test. This means that if two states are distinguishable only by features that the 1-WL test cannot separate (such as certain cycle structures or symmetries), a standard GNN will assign them the exact same embedding, and consequently, the same heuristic value, regardless of their actual distance to the goal.

This theoretical limitation was further investigated in the specific context of PDDL planning by Horčík and Šír [30], who analyzed how the choice of graph encoding affects the GNN's ability to distinguish planning states. They demonstrated that standard GNNs operating on common graph representations cannot distinguish states that are indistinguishable in C_2 logic (first-order logic with only two variables and counting quantifiers). This implies that certain symmetric structures, which are frequent in planning domains, might be conflated by the network. Consequently, the GNN may assign identical heuristic values to states that require distinct actions or have different costs-to-go, potentially hindering the search.

Building on this theoretical analysis, Horčik et al. [31] conducted a comprehensive comparative study of various state encodings for GNN-based lifted planners. They evaluated the trade-off between the theoretical expressiveness of an encoding and its practical computational cost. This analysis covers three different types of encodings:

- **object encoding**: in which the objects O are the graph vertices, and they are connected by atoms as edges.
- **atom encoding**: in which the ground atoms constitute the graph vertices, and two ground atoms are connected by the objects involved as edges.
- **object-atom encoding**, in which both objects and atoms are treated as graph vertices. This encoding creates a bipartite graph, with one partition formed by the objects and the other by the atoms.

Their results indicate that "larger" encodings, which explicitly represent every logical relation to maximize expressiveness (such as object-Atom encodings), do not necessarily lead to better planning performance. Instead, more compact

representations, specifically the object encodings, often yield superior results. Although these encodings may be theoretically less expressive, their reduced size allows for significantly faster inference times. This efficiency is critical in heuristic search, where the network must be evaluated millions of times, and suggests that the design of the state representation must carefully balance structural detail with the computational overhead of the message-passing operations.

3.2 Learning-based approaches for Numeric Planning

3.2.1 Numeric ASNets

Action Schema Networks [49] represent one of the most interesting general policy architectures for classical planning, exploiting the relational structure of PDDL action schemas to achieve strong generalization over problem size.

Recently, Wang and Thiébaux [50] introduced **Numeric Action Schema Networks** (or v-AsNets) to learn generalized policies for numeric planning. This represents the first work that exploits graph neural networks to solve numeric planning problems. This approach extends the original framework to handle the full expressivity of PDDL 2.1.

The original ASNets architecture is essentially a Graph Neural Network that alternates between two different layers: an action layer that encodes the actions via action modules, and a state layer, that encodes the boolean predicates via proposition modules. Basically, the idea is that if an action has a boolean predicate between its parameters, there will be a connection between the two modules. The *v-ASNets* architecture builds upon the existing work, adding alongside the existing modules, two novel differentiable models, designed to handle the

numeric components of the problem:

- **Fluent Modules**: In each state layer, there is one fluent module for each fluent in the problem. Fluent modules allow the network to reason directly about the quantitative components of the state space.
- **Comparison Modules**: These modules reason about the interaction between fluents, capturing their interactions in the action preconditions.

Beyond the architectural extensions, training a policy or a heuristic for numeric planning presents more difficult challenges in comparison with classical planning. Numeric problems are inherently more difficult to solve and typically entail longer plans. To mitigate the computational burden, Wang and Thiébaux [50] proposed a **Dynamic Exploration** algorithm. Unlike the standard ASNet training loop, this algorithm dynamically balances the time spent on exploration versus learning and actively manages the replay buffer to remove stale states, ensuring that the expensive teacher calls are utilized efficiently.

3.2.2 Graph Kernels for Numeric Planning

In the specific context of learning for numeric planning, the most relevant related work is the framework recently introduced by Chen et al. [11], which builds on the architecture introduced by Chen et al. [11]. Departing from the GNN paradigm, they propose a framework that uses novel graph kernels and optimization techniques tailored for numeric planning, adressing the challenge of numeric fluents and mixed attributes.

In their architecture, a state is represented as a graph where objects, numeric fluents, and goal conditions are treated as nodes, following an Object-Atom encoding adapted to numeric planning problems. Numerical information is incorporated primarily as node features, representing either the current value of a

numeric fluent or a scalar value that represents how much a numeric goal condition is violated.

This approach represents the current state-of-the-art in learning-based numeric heuristics and serves as the primary baseline for our work. It offers significant computational advantages, as kernel evaluation is typically faster than deep neural inference and the training process involves convex optimization.

However, the graph kernel representation tends to ”flatten” the structural information of numeric conditions. By treating numeric values and goal errors as scalar features rather than structural edges connecting the involved objects, this approach struggles to capture the complex relational dependencies inherent in numeric constraints.

3.3 Frameworks for Planning and Learning

3.3.1 Traditional Planning Systems

Several traditional planners have significantly contributed to the advancement of classical and numeric planning. Among the most prominent is Fast Downward [27], which remains a cornerstone in the field, offering state-of-the-art performance in classical planning via sophisticated heuristics and optimized search routines. While its core C++ implementation ensures efficiency, modifying its internal architecture or integrating custom deep learning modules often demands extensive knowledge of its codebase, creating a hurdle for rapid prototyping. Building upon this architecture, Numeric Fast Downward [2] broadened the scope of the original system to encompass numeric state variables. Both systems have been adopted in several works exploring the integration of learned heuristics [11, 9, 15], primarily because their C++ implementation allows the use of mod-

ern deep learning libraries and facilitates tight integration with neural models at the inference level. However, despite this interoperability, modifying their internal logic or extending their planning components, such as custom search algorithms or grounding procedures, requires a deep understanding of their architecture and codebase, which can be a significant barrier for rapid prototyping and experimentation, especially for researchers primarily focused on deep learning.

In the realm of numeric planning, ENHSP [42] is currently considered the gold standard. Implemented in Java, this planner is tailored to support a vast array of expressive numeric features. Although its modular structure serves as an excellent baseline, the friction associated with bridging Java applications and modern deep learning frameworks hinders its effective application in learning-augmented planning.

3.3.2 Lowering the entry barrier to planning, the Unified Planning Library

A significant milestone in democratizing AI planning is represented by the Unified Planning (UP) framework [36], a key deliverable of the AIPlan4EU project. Entirely implemented in Python, UP functions as a comprehensive middleware that standardizes interactions across a heterogeneous landscape of planning technologies. Its primary contribution lies in decoupling the high-level modeling process from the algorithmic intricacies of specific solvers. By providing a cohesive, object-oriented API, UP facilitates seamless interoperability among diverse tools, including planners, plan validators, and grounders, thereby allowing users to define and solve tasks without mastering the low-level syntax of PDDL or navigating engine-specific configurations. While this level of abstraction is highly

advantageous for educational purposes and rapid application deployment, it embodies a design philosophy distinct from ours. UP aims to simplify adoption by encapsulating the underlying complexity of the planning engines. In contrast, integrating deep learning models often necessitates complete transparency and unrestricted access to the solver's internal logic: it requires direct, granular manipulation of the search loop, node expansion, and heuristic computation—internal mechanisms that UP deliberately abstracts away to ensure a unified user experience.

Our work instead follows a different philosophy: while inspired by UP's accessibility, we aim to offer full transparency and control over each module of the framework, with a particular focus on integrating learned components. Unlike Unified Planning, which primarily provides a high-level API over existing engines, LeapNP exposes and modularizes internal components (state/action representation, search control, and learned heuristic interfaces) to enable controlled ablation and rapid prototyping.

3.3.3 Planners in Python, Pyperplan and NyX

In the domain of Python-native systems, Pyperplan [3] stands out as a lightweight STRIPS planner. Its design philosophy deliberately prioritizes code readability and architectural clarity over raw execution efficiency, positioning it primarily as a pedagogical instrument and a prototyping platform rather than a performance-oriented engine. Despite its runtime performance naturally lagging behind optimized C++ counterparts like Fast Downward, Pyperplan has found a distinct niche in learning-based research [43, 32, 12]. Its straightforward, modular Python codebase allows researchers to seamlessly inject and test neural heuristics, a task that remains cumbersome in more rigid environments.

Similarly, NyX [38] has been recently introduced as a novel planner for PDDL+

domains [17]. Built entirely in Python, it emphasizes adaptability and code comprehensibility to facilitate the rapid modeling of complex, real-world continuous processes.

Ultimately, these tools serve complementary but distinct purposes within the planning ecosystem: while Pyperplan is tailored for educational classical planning and NyX focuses on PDDL+ modeling, LeapNP is purpose-built to bridge the specific gap between numeric planning and deep learning methodologies.

3.4 Parallel Search Strategies

3.4.1 Parallel Best-First Search

A direct precursor of AWBFS (Adaptive Width Best-First Search) is **K-Best-First Search (KBFS)** [14], which generalizes Best-First Search by expanding the K best nodes from the Open List in each iteration, rather than just the single best one. While KBFS increases the likelihood of finding a better path by broadening the immediate search scope, it typically relies on a static parameter K. This introduces a rigid trade-off: a small K behaves like standard greedy search, while a large K incurs a significant computational penalty on sequential hardware. In contrast, AWBFS can be formalized as an *adaptive* KBFS where K (represented by our *width* parameter) is modulated dynamically. Our algorithm maintains $K = 1$ during heuristic descent to maximize speed and strictly increases K only when the search stagnates.

The transition from sequential to parallel search has been further explored to leverage multi-core and distributed architectures. A prominent example is **Parallel Best-NBlock-First (PBNF)** [8]. PBNF extends the standard search by selecting the best n nodes from the Open List at each iteration to be expanded in

parallel by available threads. PBNF employs a static width primarily to saturate CPU cores. More recently, Kuroiwa and Fukunaga [35] analyzed parallel GBFS using the KBFS model to identify ”pathological behaviour”, cases where parallel search expands nodes that a sequential search would ignore.

Finally, regarding distributed parallelism, **Hash Distributed A* (HDA*)** [34], works partitioning the state space across multiple processors using a hash function, where each processor manages a local Open List. While HDA* effectively scales memory and compute capacity, it incurs significant communication overhead, which becomes a bottleneck on high-latency interconnects.

3.4.2 Adaptive Beam Search

In domains such as Neural Machine Translation (NMT) and Sequence Decoding, algorithms under the umbrella of Adaptive Beam Search have been proposed to optimize the trade-off between decoding speed and accuracy. Standard Beam Search typically maintains a fixed number of hypotheses (the beam width) at each time step. However, this rigidity can be inefficient: simple inputs may require only a narrow beam, while ambiguous inputs benefit from a wider exploration.

Prominent approaches, such as the pruning strategies analyzed by Freitag and Al-Onaizan [19], introduce dynamic thresholding mechanisms. These methods effectively reduce the beam size dynamically when the leading candidate has a significantly higher probability than its competitors, or when the probability mass of the current hypotheses falls below a certain threshold relative to the best candidate. The primary goal in these contexts is *efficiency*: the adaptation is essentially ”contractive,” narrowing the search to save computation when the model is confident in its trajectory.

This stands in contrast to the objectives of satisficing planning in hard combinatorial spaces addressed in this thesis. In our context, a high confidence (or

a low heuristic value) indicates a promising descent, where a single greedy path is often sufficient. Conversely, uncertainty or lack of heuristic improvement (a plateau) signals the need for broader exploration. Consequently, while NMT approaches adapt by shrinking the beam to improve latency, our AWBFS algorithm adapts by expanding the width to improve robustness and escape search plateaus.

3.4.3 Exploration and Width-based strategies

Addressing the limitations of GBFS in search plateaus is a central challenge in satisficing planning. Common strategies include introducing randomness, as seen in Nakhost et al. [37] or Xie et al. [51], which perform random walk or local exploration to diversify the search frontier.

It is also necessary to distinguish our definition of ”width” from the concept introduced in **Iterative Width (IW)** [20]. In IW, width refers to the size of atom tuples required to define the novelty of a state. In contrast, in AWBFS, we use the term as used in the context of Beam Search literature and in KBFS [14], referring to the number of nodes expanded simultaneously.

Chapter 4

Learning Numeric Planning Heuristics with Graph Neural Networks

4.1 Architecture

Our architecture is inspired by the work of Ståhlberg et al. [45] on general policies for classical planning. We formally extend it to handle numeric planning problems, addressing two crucial challenges: (i) representing numeric fluents and conditions in a graph-based structure; (ii) integrating numeric values into node embeddings.

4.1.1 Graph Construction

Let $P = \langle D, I \rangle$ be a numeric planning problem where $D = \langle F, X, A \rangle$ is the domain and $I = \langle s_0, G, O \rangle$; let s be a state for P (a full assignments to all ground variables from X and F given O), and let $Cond(P)$ be the set of lifted conditions

for P computed joining all numeric and Boolean conditions appearing in the action preconditions and the lifting of the goals.

Let $c \in Cond(P)$, we define $ground(c, O)$ as the set of ground conditions for c obtained over the set O. For each $r \in ground(c, O)$ where list $(o_1, \ldots, o_k)$ represents the grounded objects in r, we define

$$links(r) = \begin{cases} \{(o_i, o_{i+1}) \mid o_i \neq o_{i+1},\ 1 \leq i \leq k-1\} & \textbf{if } k > 1 \\ \{(o_1, o_1)\} & \textbf{if } k = 1 \end{cases}$$

We can construct the input graph $\mathcal{G} = (V, E)$ for our GNN (that will be explained in the next section) that reflects the structure of P for a given state s as follows:

- $V = \{v_o \mid o \in O\}$
- $E = \{(v_{o_i}, v_{o_j}, r) \mid (o_i, o_j) \in links(r)\}$

Example 1 shows a graph structure constructed for a fragment of `Counters`.

Example 1. The `Counters` domain models a set of counters whose values can be controlled by two actions, respectively for increasing and decreasing. A Planning problem for this domain is one where we look for a particular configuration of such counters. Consider a problem P from the `Counters` domain. The problem contains three objects o_0, o_1, o_2. The problem includes a lifted goal condition c_3 defined as $\alpha \cdot value(?x) + \beta \cdot value(?y) + K \leq 0$. In this specific instance, the grounded constraints derived from c_3 are:
$r_{3a} : value(o_0) - value(o_1) + 1 \leq 0$ (where $\alpha = 1, \beta = -1, K = 1$)
$r_{3b} : value(o_1) - value(o_2) + 1 \leq 0$ (where $\alpha = 1, \beta = -1, K = 1$)
From this, $links(r)$ will be: $links(r_{3a}) = [(o_0, o_1)]$, $links(r_{3b}) = [(o_1, o_2)]$. Which in our graph will translate to edges $(v_{o_0}, v_{o_1}, r_{3a}), (v_{o_1}, v_{o_2}, r_{3b})$.

Intuitively, the graph formalises how objects are related among each other. It does so by looking at the preconditions and goals, and in particular at those that are satisfied by the state at hand. This piece of information are meant to capture relevant parts of the state space.

Now that we have the graph formalised, we detail how to construct a GNN that use the above graph as a skeleton for the inference task.

4.1.2 GNN Construction

When we construct the GNN, each node in the input graph will have an associated node embedding of size k, where k is the number of features for a node; k is an hyperparameter of our network. Overall, we will have a set of node features $Z \in \mathbb{R}^{k \times |V|}$, each of which represents a node in V.

For each condition $c \in Cond(P)$, we create two $MLPs$. If c is a Boolean condition, we follow Ståhlberg et al. [46]: one MLP, namely $\top$-MLP$_c$ is devoted to collect information from those objects involved in groudings of c true in the current state (thus encoding the current situation); the other one, i.e., G-MLP$_c$ processes those that appear in the goal specification (thus encoding the target).

This idea does not extend directly to numeric planning as (i) numeric preconditions and numeric goals can have a very different structure, (ii) each numeric condition can be (un)satisfied by a state with a different degree. For this reason we first distinguish whether a numeric condition is involved in a goal (a) or in a precondition (b). For case (a), we create two MLP. The former, namely G-MLP$_c$ collects information by all objects belonging to groundings of c (i.e., the target), while the second $\top$-MLP$_c$ focuses on those linked to condition satisfied in the current state (i.e., the current situation). For numeric preconditions, we take a different approach. That is, we associate one MLP to the edges induced by groundings that are satisfied, namely $\top$-MLP$_c$, in the current state and another

to those that are not, namely $\bot$-MLP$_c$. This choice ensures that (i) there is always an MLP that handles the objects involved in a grounded numeric precondition, regardless of its truth value, and (ii) the network can learn different weights for satisfied and unsatisfied cases.

We label numeric preconditions and numeric goals differently so that the MLPs can be shared among conditions in preconditions only (having identical numeric goals does not make much sense). That is, if we have two identical conditions c_0 and c_1 where c_0 is a lifting of a goal condition and c_1 is a lifted precondition, they will have different associated MLP. Differently, if both c_0 and c_1 are lifted preconditions, they will share the same MLPs. This may happen in cases where different actions have some common requirements for execution.

The particular structure of the MLP is yet another parameter of the architecture, whose input and output are defined as below.

We distinguish the reasons why two objects are related using the edges in E and the current state s. This way, we know which MLP needs to collect the embeddings of the objects expressed in the edge. For instance if $(o_1, o_2, r) \in E$ is such that $c \in C$ and c is a lifting of r, and $r \models s$, then we connect these objects to $\top$-MLP$_c$. Similar for the other cases. In the following we detail how an MLP collects information, focusing on the case in which c is a numeric condition.

For each $(v_{o_i}, v_{o_j}, r) \in E$ with r being a grounding of c we create a number of inputs for the associated MLP_c; more specifically the MLP_c takes the embeddings of the pair v_{o_i}, v_{o_j}, i.e., $\mathbf{h}(v_{o_i})$ and $\mathbf{h}(v_{o_j})$. Then, we also add all numeric values x mentioned in r that are associated with the specific pair (v_{o_i}, v_{o_j}), considering state s. Those include the evaluation of the numeric fluents mentioned in r, the coefficients associated with each numeric fluent, and the constant value in r. Note that, for a Boolean condition c, the related MLP_c takes as input only the node embeddings; Boolean conditions can be considered as a special case in

which there are no numeric values involved. The resulting input for the MLP_c will be a matrix of dimensions $(m, k+n)$ if the condition is unary, or $(m, 2k+n)$ otherwise, with m being the number of pairs in input, n being the number of numeric values involved in r and associated with the specific pair, and k being the embedding dimension.

We denote with $input(MLP_c)$ all nodes used as an input of the MLP_c.

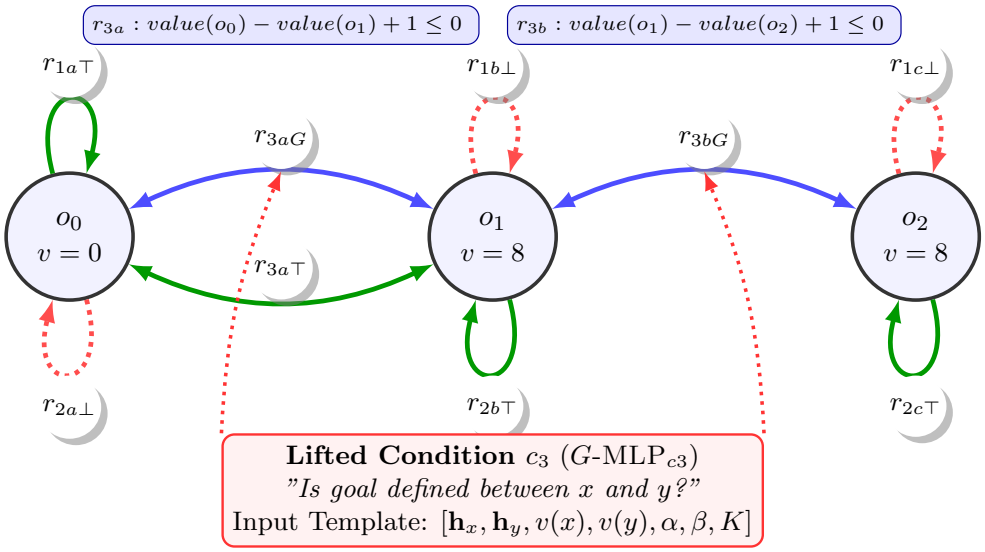
Then each MLP_c generates, for each input node, a message embedded in a vector of size k. More precisely, at each iteration l of the GNN, the message m_c outputted by the MLP_c is as follows:

$$\mathbf{m}_c = \text{MLP}_c\left([\mathbf{h}_l(v_{o_i}), \mathbf{h}_l(v_{o_j}), x_1, \ldots, x_n]_{p=1}^{m}\right) \tag{4.1}$$

- each $\mathbf{h}$ is indexed with the iteration number l;
- $\mathbf{h}_l(v_{o_i}), \mathbf{h}_l(v_{o_j})$ are the object embeddings for one row of the input matrix;
- m and n are the number of pairs and the number of numeric values for each pair given as an input, respectively;
- m_c is a matrix where each row represents the message for a node in V. We denote with $m_{v,c}$ the row containing the message for node v.

Example 2. Figure 4.1 illustrates the graph architecture and the feature extraction process for a problem P from the `Counters` domain. The problem contains three objects o_0, o_1, o_2 with current numeric values $value(o_0) = 0$, $value(o_1) = 8$, and $value(o_2) = 8$. The problem includes two lifted numeric preconditions c_1 and c_2, and a lifted goal condition c_3 defined as:

- $c_1 : value(?c) - max\_int + K \leq 0$
- $c_2 : K - value(?c) \leq 0$



Type	Symbol	Meaning
Lifted	c_1	$value(?o) - max\_int + K \leq 0$
Lifted	c_2	$K - value(?o) \leq 0$
Lifted	c_3	$\alpha \cdot value(?o_i) + \beta \cdot value(?o_j) + K \leq 0$
Ground	r_{1a}, r_{1b}, r_{1c}	Specific Instances of Precondition c_1
Ground	r_{2a}, r_{2b}, r_{2c}	Specific Instances of Precondition c_2
Ground	r_{3a}, r_{3b}	Specific Instance of Goal condition c_3
Visual State Mapping:		
→		Green: Condition Satisfied ($r \in ground(c, O)$ is True)
⇢		Red Dotted: Condition Unsatisfied ($r \in ground(c, O)$ is False)
↔		Blue: Structural Definition ($r \in ground(c, O)$ exists)

Figure 4.1: Graph architecture example

- $c_3 : \alpha \cdot value(?o_i) + \beta \cdot value(?o_j) + K \leq 0$

The visual representation maps the logical state of conditions to the graph:

- **Green edges** represent satisfied conditions ($r \in ground(c, S)$ is True).
- **Red dotted edges** represent unsatisfied preconditions ($r \in ground(c, S)$ is False).
- **Blue edges** represent the structural definition of the goals.

Focusing on the edge r_{3bG} connecting o_1 and o_2, the associated G-MLP$_{c3}$ receives a feature vector constructed from the Input Template $[h_x, h_y, v(x), v(y), \alpha, \beta, K]$. Given the current state values $value(o_1) = 8, value(o_2) = 8$ and the constraint coefficients defined in c_3 ($\alpha = 1, \beta = -1, K = 1$), the specific input row for this edge is:

$$[\mathbf{h}_l(o_1), \mathbf{h}_l(o_2), 8, 8, 1, -1, 1]$$

where $\mathbf{h}_l$ represents the latent embeddings of the object nodes.

As for any GNN inference task, we need to have a way to aggregate all messages directed to a specific object, so that the embedding of that object is changed accordingly. So, let $input(MLP_c)$ be the set of nodes in input for the MLP_c, we define $\mathbf{a}^v_{(l)}$ as the aggregation of all messages for v, formally computed as follows:

$$\mathbf{a}^v_{(l)} = \text{AGG}\big(\{m_{v,c} \mid \exists MLP_c : v \in input(MLP_c)\}\big) \tag{4.2}$$

The aggregation function is a hyperparameter of the architecture determining how information from different relations is combined. Typical choices are sum, mean, or smooth-max.

After aggregation, we update the embedding of each node v using the aggregated vector $\mathbf{a}^v_{(l)}$. The update function takes as input the embedding of the node

at layer l, and the aggregated vector, and gives as output the embedding of the node at the next layer.

$$\mathbf{h}_{l+1}(v) = \text{UPDATE}\left(\mathbf{h}_l(v), \mathbf{a}^v_{(l)}\right) \tag{4.3}$$

As update function we use an MLP, i.e., MLP_U, which is the same for each node in the GNN.

After L layers of propagation, with L being a hyperparameter, the final embeddings $h_L(v)$ are given to a readout function, whose output is our heuristic value $h(s)$. The readout function uses two different $MLPs$, MLP_1 and MLP_2, :

$$h(s) = \text{MLP}_2\left(\sum_{v \in V} \text{MLP}_1(\mathbf{h}_L(v))\right) \tag{4.4}$$

4.1.3 Generalization

Our scope is to learn a heuristic function from some problems and then use that knowledge to solve other problems in the same domain. More specifically we aim to learn over smaller instances, and then use that knowledge to solve larger instances. We do so through the following workflow. We start from some instances set, create the associated GNN with the method described above, and then optmise the weights of our GNN so that they minimise a loss function that given a state s and goal, takes as an input the optimal distance $h^*(s)$, and the distance predicted by the GNN, i.e., $h(s)$. Then, we can use the given weights to instantiate the weights of some other GNN constructed for other problems inside the domain. We can do so because our GNNs are associated with the lifted representation of the domain, i.e., $Cond(P)$. The nodes in our input graph are the objects, that are specific for some instance, while the edges depends on the state s, what we can learn and use to generalize to all the instances in a given domain are the lifted conditions specific to the domain, i.e, the set $Cond(P)$.

Each of these conditions has an associated MLP_c. The weights associated with each MLP_c, along with the weights associated to MLP_U, responsible for the UPDATE function, and the weights associated to MLP_1 and MLP_2, responsible for the READOUT function, constitutes the knowledge that we extract to solve unseen instances.

A complexity that arises in numeric planning problems is that the language of numeric conditions is not bounded. This means that, while the lifted preconditions of the actions are fixed, the numeric goals are not. For this reason, we have to understand which problem inside a domain can still use the weights learned from another problem inside that domain. We do so by reasoning on the structure of the problem given in terms of the $Cond(P)$, and in particular on the lifted fluents list associated with each goal condition in $Cond(P)$.

More precisely, let $fluents(c)$ be the list of fluents involved in a goal condition c, i.e., $fluents(c) = [f_1, \cdots, f_t]$ such that $f_1, \cdots, f_t \in c$ and the order satisfies the order provided by c, a problem P can exploit the knowledge acquired during training over problems $T = \{P_1, \cdots, P_m\}$ if an only if, the set of fluents lists associated to the goal conditions of a problem is a subset of all fluents lists associated to the goal conditions of the training problems, i.e., let $Goals(P) \subseteq Cond(P)$ be the subset of $Cond(P)$ that contains only the goal conditions, for each $c \in Goals(P)$ there exists a $P' \in T$ and $c' \in Goals(P')$ such that $fluents(c') = fluents(c)$.

To understand why, observe that the weights of our GNN solely depends on the relationships of the fluents within the lifted conditions they are involved in.

This means that in order to generalise to different problems, it is necessary that for each lifted goal condition inside the problem, it has already been considered (in terms of relationship) during training. For instance, if we happen to solve a problem with a goal $position(?x) + position(?y) \geq 0$, we need to have in our

training set another problem where the fluents list of such a goal has appeared in the goal.

Example 3. Let's say that we want to solve a new instance of counters with a goal that is $2 \cdot value(o_1) + 3 \cdot value(o_2) \leq 5$, the numeric lifted fluents related to this goal are $value(?o_i)$ and $value(?o_j)$. From the previous example, we have a lifted goal in the training set that is $\alpha_1 \cdot value(?o_i) + \alpha_2 \cdot value(?o_j) + \beta \leq 0$. The set of numeric fluents for this goal is $(value(?o_i), value(?o_j))$, therefore, since the set of numeric fluents is the same required by the new goal condition, we can generalize to the new problem.

4.1.4 State Encodings and Heuristic Computation

Algorithm 1 summarizes the message-passing process and heuristic value computation.

The input for a state s is the set of nodes V, the set of edges E, and the set of lifted conditions for the planning problem $Cond(P)$. L and k are hyperparameters describing the number of layers and the embedding dimension, respectively.

Lines 1–5 handle the initialization of the nodes. The first part of the embedding is initialized with zeros, while the second part is randomized. The reason behind this partial randomization derives from [1], where it is shown that: (i) a randomization in the embedding vector can greatly enhance the expressiveness of the GNN; (ii) for datasets with varying expressiveness requirements (as in our case), the best performances are achieved with a partial randomization. Lines 7–8 concern the message-passing operation. At each layer l, the algorithm iterates over the set of lifted conditions $c \in Cond(P)$. For each condition c, we identify the subset of edges $E_c \subseteq E$ corresponding to the groundings of c present in the graph. Specifically, let $e_r = \{(v_{oi}, v_{oj}, r) \in E | r \text{ is a grounding of } c\}$. The

Algorithm 1 GNN mapping a state s to heuristic value $h(s)$

Input: $\langle V, E, Cond(P)\rangle$. **Output**: $h(s)$.

1: **for** each $v \in V$ **do** ▷ Initialize encoding values
2: **for** $i \in (0, k/2)$ **do**
3: $\mathbf{h}_0(v)[i] = 0$
4: **for** $i \in (k/2, k)$ **do**
5: $\mathbf{h}_0(v)[i] = random(0, 1)$
6: **for** $l = 0$ to $L - 1$ **do**
7: **for** each $c \in Cond(P)$ **do**
8: $\mathbf{m}_c = \text{MLP}_c\left([\mathbf{h}_l(v_{o_i}), \mathbf{h}_l(v_{o_j}), x_1, \ldots, x_n]_{p=1}^{m}\right)$
9: **for** each $v \in V$ **do**
10: $\mathbf{a}_{(l)}^{v} = \text{AGG}\big(\{m_{v,c} \mid \exists MLP_c : v \in input(MLP_c)\}\big)$
11: $\mathbf{h}_{l+1}(v) = MLP_U(\mathbf{h}_l(v), \mathbf{a}_{(l)}^{v})$
12: $h(s) = MLP_2(\sum_o MLP_1(\mathbf{h}_L(v)))$

associated $MLPc$ processes all these edges in a single batch. It takes as input a matrix constructed from the embeddings of the node pairs defined in E_c $(i.e., h_l(v_{oi}), h_l(v_{oj}))$ and their associated numeric values $(x_1, \ldots, x_n)$. This explicitly links the graph topology defined by E to the inference logic: the MLP for condition c only processes the pairs of objects actually connected by a relation of type c in the current state.

Lines 9–11 handle the aggregation and update steps. The algorithm iterates over each node $v \in V$ to collect the incoming messages. Line 10 corresponds to the aggregation function, in which the messages $m_{v,c}$ relevant to node v are aggregated into a single vector of size k. The aggregation function is a hyperparameter. Line 11 regards the UPDATE function, where each node is updated

with the aggregated message.

In line 12, the final embeddings of each node are passed to the first MLP, summed together, and then passed to a second MLP to extract the heuristic value. All MLPs consist of an input layer with a linear activation function, a dense layer with ReLU activation function, and a dense layer with a linear activation function.

4.1.5 Training Procedure

The objective of the training phase is to optimize the parameters θ of the Graph Neural Network so that the predicted heuristic value $h_\theta(s)$ for a state s approximates the optimal cost $h^*(s)$ as closely as possible.

Data Generation The training dataset is composed of pairs $\langle s, y \rangle$, where s is a valid state reachable from the initial state s_0, and y is the target value representing the optimal distance to the goal. As detailed later in Section 5.1, the target values are generated using an optimal numeric planner.

Loss Function To train the network, we formulate the problem as a regression task. We employ the Mean Squared Error (MSE) as the loss function $\mathcal{L}$, which measures the average squared difference between the estimated heuristic values and the actual optimal costs. Formally, given a batch of N training samples, the loss is defined as:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(h_\theta(s_i) - y_i \right)^2$$

where:

- N is the number of samples in the batch;

- $h_\theta(s_i)$ is the heuristic value predicted by the GNN for state s_i given parameters θ;
- y_i is the optimal cost for state s_i.

Optimization and Hyperparameters The minimization of the loss function is performed using the Adam optimizer; the training process and the resulting models are governed by the following hyperparameters:

- **Learning Rate:** We utilize a fixed learning rate of $\eta = 0.0002$.
- **Number of Layers (L):** The GNN performs $L = 30$ message-passing iterations. This depth allows the network to propagate information across long-range dependencies in the graph.
- **Embedding Size (k):** The size of the node embeddings is set to $k = 60$.

Aggregation Function While previous works in classical planning often employed smooth-max aggregation, we empirically found that the add-aggregation function yields superior performance for capturing numeric features. What we observed is that the smooth-max aggregation function has the tendency to cancel the numeric part of our representation. The add and smooth-max aggregation functions are respectively formalized in Hamilton [24] and Ståhlberg et al. [46]. We also tried two other aggregation functions, namely mean and max aggregation (explained in Hamilton [24]), but they did not yield any positive result.

Stopping Criteria The training procedure goes for up to 1000 epochs. To prevent overfitting, we employ an early stopping strategy that terminates the training procedure early if the validation error does not improve after 30 epochs. For

almost every domain, the training stops before reaching 100 epochs, with the only exception of `Hydropower` in which the training stops around epoch 150.

4.2 Numeric Planning in Python: LeapNP

Different works have shown how deep learning approaches can be used to solve classic and numeric planning problems, either as heuristic functions inside a planner or as value functions for general policies. When it comes to heuristic functions, integrating learning-based heuristics inside a planner can be challenging due to the complexity and rigidity of many existing planning systems. These systems are often written in low-level languages and optimized for performance, making them difficult to modify or extend with machine learning components; this limits the ability to prototype and test new ideas efficiently. To address this, the new heuristic has been implemented in a new framework called **LeapNP**[7], which stands for Learning and Planning Framework for Numeric Problems. The framework is designed to be accessible and easily extensible, in order to lower the technical barriers typically associated with integrating deep learning and automated planning.

4.2.1 Components

In this section, we will present and discuss the different components that constitute the architecture of LeapNP. Except for the parser, the components can be changed without impacting the others, leading to an architecture that is simple to understand and highly customizable. The diagram in Figure 4.2 provides an overview of the high-level modules that compose the framework.

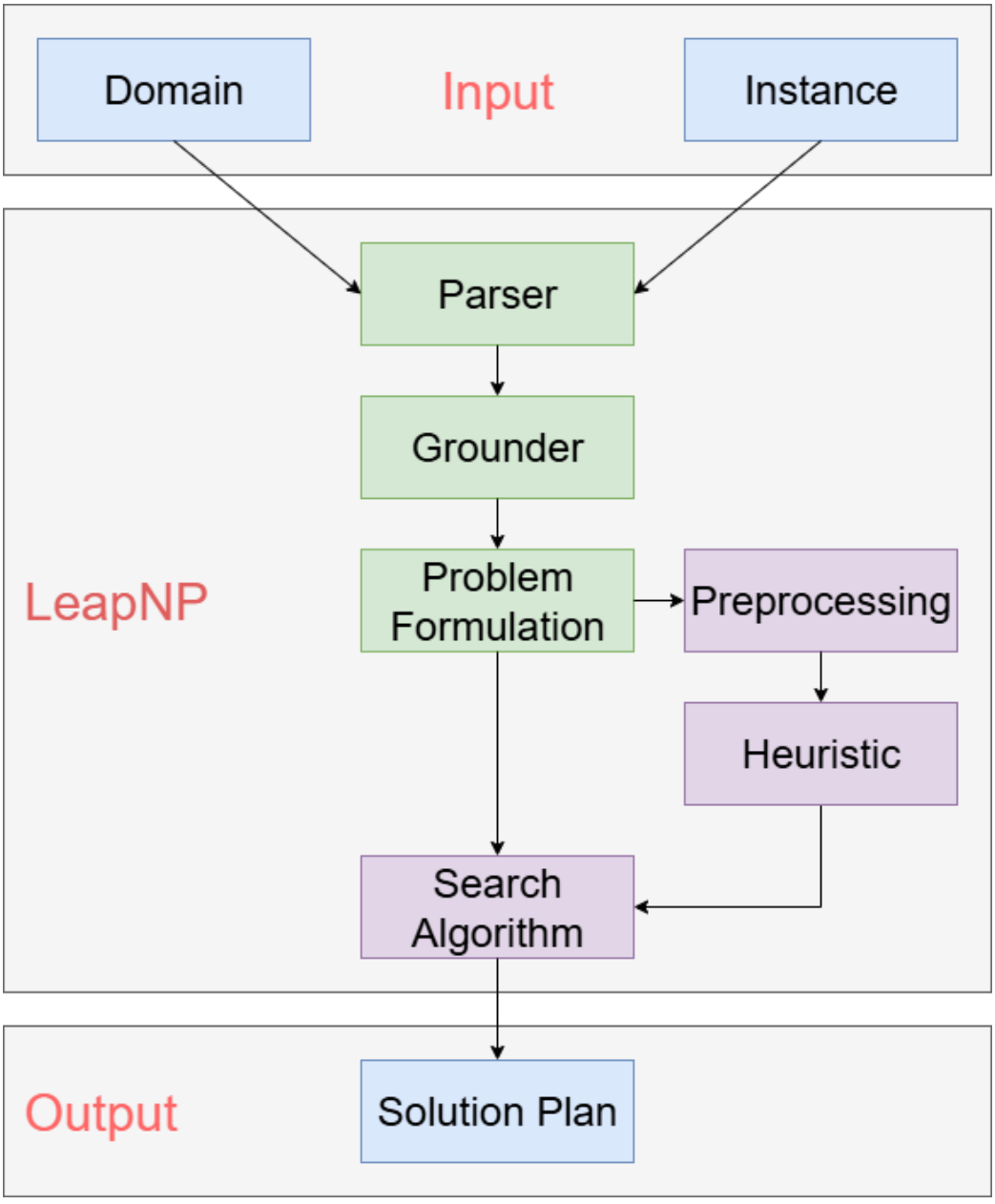


Figure 4.2: Visual representation of the components that constitute our architecture.

Parser and Grounder The parser uses the PDDLReader defined in the Unified Planning framework [36], which reads the problem in PDDL format and parses it into a Unified Planning (UP) problem. Alternatively, it is also possible to use the problem directly in the UP format as input. The parser supports classic, numeric, and temporal planning problems. The grounder takes the parsed problem and grounds it (Figure 4.2), generating the grounded actions from the action schema and the objects of the problem. As a grounder, we use an implementation of the grounder used in the ENHSP planning system (`https://github.com/hstairs/jpddlplus`), along with its adaptation to the UP framework, which gives us the grounded problem and the traceback map between the grounded actions and the action schemas; moreover, the grounder can be simply changed to use every grounder implemented in the UP framework. If a custom grounder is needed, it can be integrated by implementing it inside the UP framework; the UP documentation provides a guided procedure to support this process.

Problem Formulation The Unified Planning framework has been created as a wrapper around existing planners, grounders, and other planning components. It is effective in parsing the problems, and it permits us to use different interchangeable grounders. The issue is that UP data structures are designed to maximize interoperability among different planning techniques and to facilitate problem manipulation, rather than being optimized for fast search or to offer intuitive representations. To better support our use case, once the problem has been grounded, we encode the relevant components, namely the initial state of the problem, the goals, and the grounded actions, into custom data structures that are more lightweight and easier to manipulate within our framework.

Preprocessing and Heuristic Since different heuristics may rely on distinct preprocessing steps, our framework allows for flexible initialization. To implement a new heuristic, only two components are needed: an initialization routine (or preprocessing) to store any precomputed structure, and an evaluation function that maps a given state to a heuristic value for the search algorithm. The latter is the function that will be later used by the search algorithm.

Search Algorithm The search algorithm takes two inputs: the planning problem obtained by the problem formulation module, and the heuristic that it will call to evaluate new states (Figure 4.2). New search algorithms can be written, simply extending the Search Algorithm base class and implementing the new algorithm in the solve function. If a solution is found, the algorithm returns a solution plan for the given problem.

4.2.2 Modules

To encode the planning problem from the UP structures, we designed different modules to handle states, actions, and goals, focusing on enabling a simple and intuitive representation of numeric planning problems as well. The modules involved in this formulation are shown in Figure 4.3, which provides an overview of the data structures used to represent the core components of a planning instance.

Object and Atom The `Object` module serves as the foundational representation for problem objects; it simply defines the name and the type of each object, which are later used in the construction of atoms, actions, and state representations.

Atoms represent the basic grounded fluents of the domain, and are used in

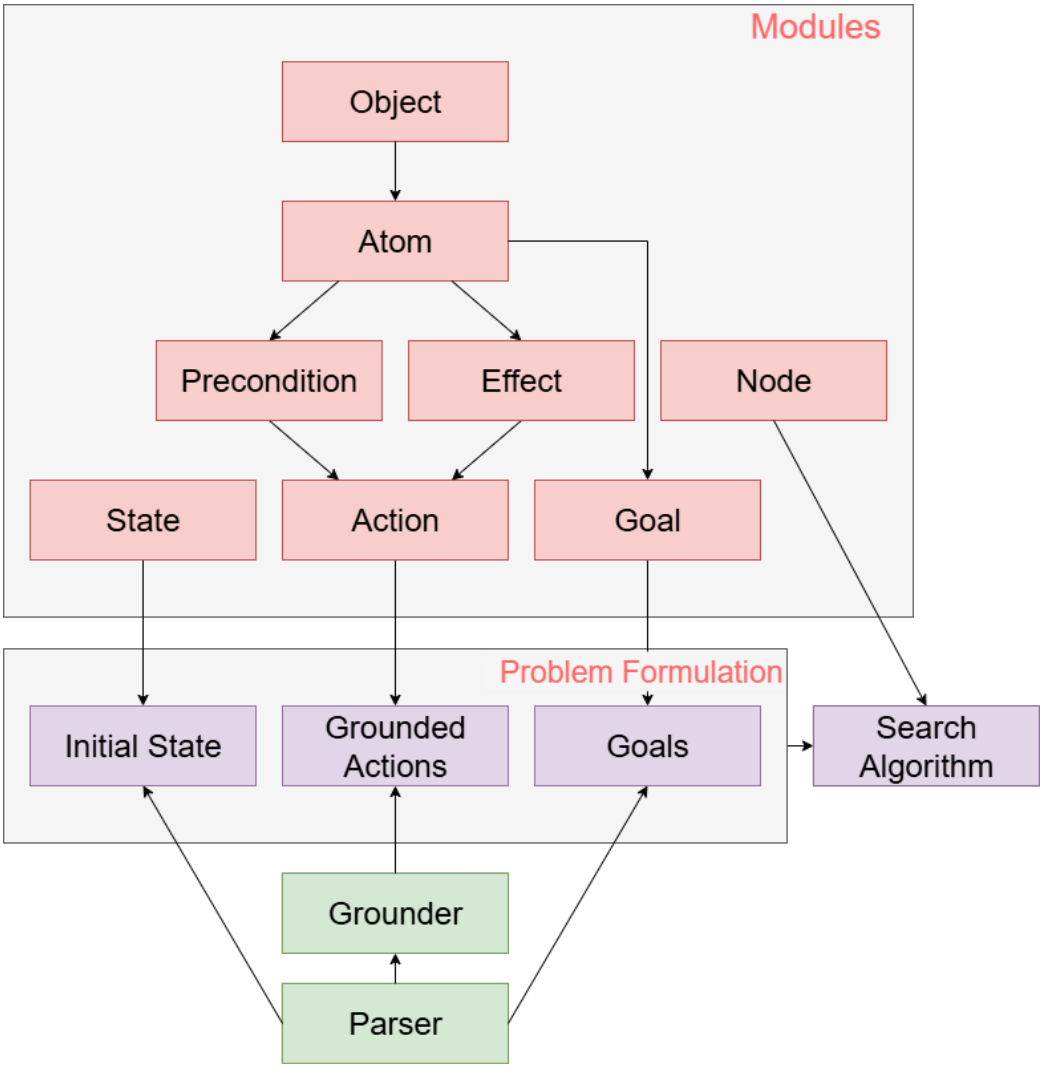


Figure 4.3: Visual representation of the data modules used in the problem formulation.

preconditions, effects, and goals. They provide a unified interface to access and manipulate variable values within expressions. Each atom object includes:

- The name of the atom.
- The list of objects it involves.
- The name of the corresponding lifted fluent.

For numeric reasoning, expressions are parsed as strings and evaluated dynamically: for each atom referenced in an expression, its name is replaced with its value in the current state, and the resulting expression is computed using a lightweight mechanism. This approach, while simple, enables flexible and expressive modeling of numeric conditions and updates.

State and Node representation A state represents a configuration of the world at a given moment. It is defined solely by the current values of the fluents (both boolean and numeric) and describes the situation in which the planning problem currently stands. A state does not contain any additional information about how it was reached or the cost associated with it. A node instead is a data structure used within the search algorithm to manage and track planning information. A node encapsulates a state but also includes additional data. Each node object includes:

- A dictionary mapping variable names to their current values (both boolean and numeric).
- A unique identifier derived from a hash of the node's variable values, used to avoid revisiting duplicates.
- The g value (cost from the initial state to the current state).

- The h value (heuristic estimate to the goal).
- A reference to the parent node.
- The action that generated the node from its parent.

The `Node` module provides:

- Comparison and hashing utilities to manage the priority queue and the $gMap$.
- A method to expand the node by applying all applicable grounded actions, generating valid successors.

Action Encoding The `Action` module represents grounded actions, each derived from an action schema instantiated with concrete objects. Actions include both preconditions and effects, which can be either boolean or numeric. Each action object stores:

- A unique name (e.g., `increment_c1`).
- A list of preconditions.
- A list of effects.

Preconditions and effects are represented using their own dedicated modules:
Precondition:

- A name for the condition.
- A list of atoms (fluents) referenced.
- A flag indicating whether the condition is boolean or numeric.

- A method to check satisfaction of the condition in a given state.

Effect:

- A name for the effect.
- A list of involved atoms.
- A flag indicating whether the effect is boolean or numeric.
- A left-hand side (a fluent).
- A right-hand side (a constant or expression).
- An operator (e.g., $=, +=, -=$) for numeric effects.
- A method to apply the effect to a given state.

Example 4 (Counters domain)**.** The objective of this numeric planning domain is to increase or decrease the value of the counters to achieve a particular configuration of such counters.
Current State: $\texttt{value}(c_0) = 0, \texttt{value}(c_1) = 8,\ \texttt{max\_int} = 8.$
Goals: $\texttt{value}(c_0) + 1 \leq \texttt{value}(c_1)$
Atoms: $\texttt{value}(c_0), \texttt{value}(c_1).$
Constants: $\texttt{max\_int}.$
Objects: $\texttt{counter}\ c_0, c_1.$
Grounded Actions: $\texttt{increment\_}c_0, \texttt{decrement\_}c_0, \texttt{increment\_}c_1, \texttt{decrement\_}c_1.$
Action Preconditions: $\texttt{value}(?c) - \texttt{max\_int} + 1 \leq 0, 1 - \texttt{value}(?c) \leq 0.$
Action Effects: $\texttt{value}(?c) = \texttt{value}(?c) + 1, \texttt{value}(?c) = \texttt{value}(?c) - 1.$

When the search algorithm expands a new state, it first check, for each grounded action, if the action is applicable in the current state. For the action $\texttt{increment\_}c_0$

the only precondition is $\texttt{value}(c_0) - \texttt{max\_int} + 1 \leq 0$. c_0 is a constant; therefore its value is already in the precondition, which becomes $\texttt{value}(c_0) - 8 + 1 \leq 0$, then, to evaluate if the precondition holds it checks the list of atoms for the precondition, in this case only $\texttt{value}(c_0)$, it extract its value from the current state, and the precondition becomes $0 - 8 + 1 \leq 0$, then, the precondition is evaluated, and if it holds, the same process occurs for the action effects. The only effect of this action is $\texttt{value}(c_0) + 1$, which again is evaluated, and the result is stored in the new state.

Goal Conditions The `Goal` module defines the desired final conditions for the planning task. These are evaluated over a candidate state to determine whether it satisfies the problem objective.

Each goal consists of:

- A string representing the goal expression.
- A list of referenced atoms.
- A boolean flag distinguishing between boolean and numeric goals.
- A method to verify goal satisfaction over a given state.

4.2.3 Search Algorithms

All the Search algorithms extend the base SearchAlgorithm class, which consists of two variables, representing the number of expanded states and the number of evaluated states, utility functions to update those parameters, a function to extract the plan and the goal state once a goal is reached, and a solve function that must be implemented when extending the base class. When creating a new search algorithm, we also extend the base Node class by defining a new Node

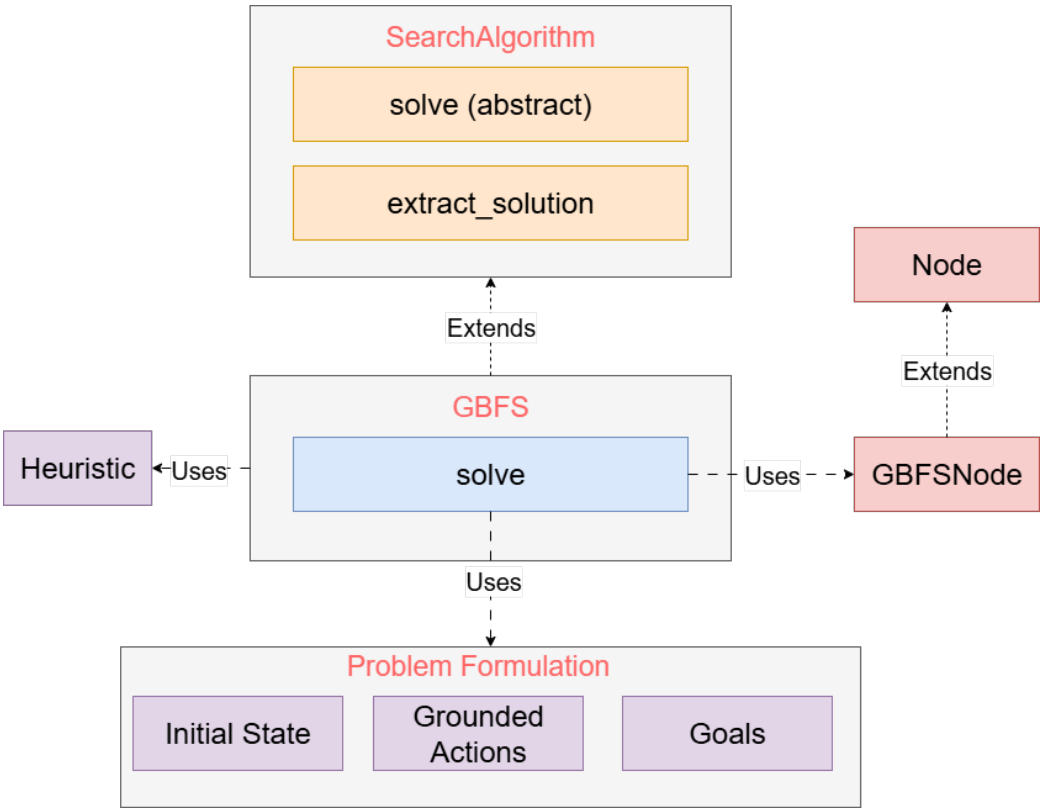


Figure 4.4: Visual representation of the structure used for the implementation of the GBFS search algorithm, utility functions are omitted for simplicity.

class specific to the algorithm. This is necessary because the Node class includes functions used to order the nodes in the priority queue and the function responsible for generating the neighborhood of the current node. An example for the GBFS search algorithm can be seen in Figure 4.4. Finally, for each implemented search algorithm, we have also developed a variant that we refer to as *multiple evaluation*. The key difference lies in the way successor states are evaluated: instead of evaluating them one by one as they are generated, all successor states are evaluated in parallel. While this approach does not offer significant advantages when using traditional heuristics, it becomes highly beneficial when employing

heuristics based on deep learning methods. If the methods used support parallel evaluation, this strategy can significantly reduce the overall search time. Traditional search algorithms (Greedy Best-First Search [6], A* [25] and WA* [13]) and their multiple evaluation variants are implemented using two different data structures to store the states: states that have not been expanded are stored in a priority queue, and are ordered following the comparison key defined when extending the Node base class into the Node specific for the search algorithm. For example, for Greedy Best-First Search (GBFS) [6], states are ordered following the h value, from smallest to highest. The second data structure is a map between the states and their cost g, which we will call $gMap$, which keeps track of already seen states for two reasons: to avoid re-expanding the same state multiple times, and to keep track of the parent and the action that led to a state, in order to reconstruct the plan once the goal state is found. To avoid re-evaluating the same state multiple times, when a state is evaluated, it is also added to the $gMap$, and if the same state is encountered again during the search, it is added to the priority queue only if its g value is lower than the one already present in the $gMap$.

A* The A* algorithm [25] is widely used for its ability to find optimal solutions, provided that the heuristic function is admissible. In our framework, we implemented A* by extending the base SearchAlgorithm class. The corresponding node implementation overrides the comparison method to order the priority queue based on the f-value, defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost accumulated from the initial state to the node n, and $h(n)$ is the heuristic estimate to the goal. While the optimality of A* is guaranteed only with admissible heuristics, a property that learning-based heuristics usually do not satisfy, the algorithm remains an integral part of the library, catering to scenarios where min-

imizing plan cost is prioritized over runtime performance, or where the learned heuristic is specifically trained to approximate optimal values.

We also included Weighted A* (WA*) [13]. This variant modifies the evaluation function to $f(n) = g(n) + w \cdot h(n)$, with $w > 1$. By inflating the heuristic influence, this parameter makes the search greedier, prioritizing nodes that appear closer to the goal over those with lower accumulated costs. This introduces a trade-off: it typically reduces the number of node expansions and runtime significantly, at the expense of solution optimality.

Greedy Best-First Search Greedy Best-First Search (GBFS) [6] is a search strategy that prioritizes exploration based solely on the heuristic value $h(n)$. This approach typically leads to faster solutions compared to A*, albeit without guarantess on optimality, and in general with a worse quality on the found plans. The implementation with our modules of the algorithm can be seen in Figure 4.4. GBFS serves as the primary baseline for our experimental analysis, as our proposed variants are essentially parallelized extensions of this standard greedy approach.

Parallelism over different states The key idea behind our multiple evaluation variant is quite simple: deep learning methods are traditionally capable of parallelizing easily between input instances. However, with GNNs, this parallelization is usually less trivial. In GNNs both vertices and edges are connected: vertices are connected with edges, and edges share common vertices. Therefore, parallelizing over one state is not a trivial matter. What we can do instead is give as input multiple states: each state represents a graph that is independent from the others, achieving something similar to mini-batch parallelism (refer to [4] for additional information on this matter). In addition, one issue with mini-

batch parallelism is the possibility of causing load imbalance when graphs have large differences in their dimension. In our case, our graphs are quite similar in their dimensions: the number of nodes represents the objects of the problem, and the different relations are fixed for the domain. There will be a difference in the number of arches among different instances, but not enough to cause a load imbalance.

Note that this type of parallelization is not new; it is a technique often used in general policies, in [46]. While not explicitly stated, it is used in their architecture. Our contribution relies on the idea of introducing this parallelism inside traditional search algorithms, such as GBFS [6], A* [25], and WA* [13], enabling more efficient exploration by evaluating multiple states simultaneously through a single GNN forward pass. This approach not only improves computational efficiency but also opens the door to integrating learned policies more tightly with our framework. We refer to these parallelized implementations as MBFS, MA*, and MWA*.

Deferred multiple evaluation In general policies, the search algorithm does not keep track of evaluated states; at each step, it selects the best successor state and discards the rest. Therefore, further parallelization beyond the current state would not be beneficial. In our case, however, we can extend the parallelization beyond the immediate successors of a single state. This is possible because our algorithm retains and evaluates multiple candidate states concurrently. By deferring the evaluation of successor states and batching them across multiple parent nodes, we achieve deeper parallelization that further leverages the computational capabilities of modern hardware.

The complete pseudocode is shown in Algorithm 2. The inputs are the initial state of the problem s_0, the set of grounded actions A, the goal of the problem

Algorithm 2 Deferred Multiple Evaluation Best-First Search (DBFS)

Input: $\langle s_0, A, G \rangle$ — initial state, list of grounded actions, and goal.

h — heuristic function.

Output: Solution plan from initial state to goal, or none.

$Open \leftarrow \{s_0\}$ ▷ priority queue ordered by h

$gMap[s_0] \leftarrow \{0\}$ ▷ Maps state to best known g-value

while $Open \neq \emptyset$ **do**

  $s \leftarrow$ Pop($Open$) ▷ state with lowest h

  if IsGoal(s, G) **then**

    return ExtractSolution($s, gMap$)

  $Batch \leftarrow \emptyset$

  for all $s' \in$ Successors(s, A) **do**

    if $s' \notin gMap \vee gMap[s] + c(a, s) < gMap[s']$ **then**

      $gMap[s'] \leftarrow \{(gMap[s] + c(a, s))\}$; $Batch \leftarrow Batch \cup \{s'\}$

  $NewBatch \leftarrow \emptyset$

  for all $s' \in Batch$ **do**

    for all $s'' \in$ Successors(s', A) **do**

      if $s'' \notin gMap \vee gMap[s'] + c(a, s') < gMap[s'']$ **then**

        $gMap[s''] \leftarrow \{gMap[s'] + c(a, s')\}$; $NewBatch \leftarrow NewBatch \cup \{s''\}$

  $Batch \leftarrow Batch \cup NewBatch$

  ParallelEvaluate($Batch, h$) ▷ single GPU call

  $Open \leftarrow Open \cup Batch$

return NONE

G, and the heuristic function h. After the initialization (lines 1-2) and a goal satisfaction check (lines 4-6), the core logic diverges from standard GBFS in the expansion phase. Instead of evaluating successors immediately, the algorithm employs a two-stage collection process. First, the immediate successors of the current state s are generated and added to a temporary *Batch* container without being evaluated (lines 7-10). Subsequently, the algorithm iterates through these unevaluated successors to generate a second layer of states, which are collected in NewBatch (lines 11-15). In both cases, a state is added only if it has not been evaluated before (i.e., it must not be in the $gMap$), or if it has been evaluated with a higher accumulated cost g. Finally, the two sets are merged, and the heuristic values for all collected states are computed in a single *ParallelEvaluate* call, before putting them in the *Open* list (i.e., the priority queue) to order them(lines 16-18).

This idea is inspired by deferred heuristic evaluation, introduced in Fast Downward [27] as a variant of best-first search. In that setting, successor states are not immediately evaluated; instead, they are inserted into the open list with the heuristic value of their parent. The original goal of this technique was to reduce the number of heuristic evaluations, thereby improving efficiency. In our case, we repurpose this concept with the opposite objective: we aim to maximize the number of evaluations performed in parallel. By deferring individual evaluations and accumulating a larger batch of successors across different parents, we enable highly efficient, parallel GNN-based inference. This strategy allows us to process a broader set of promising states simultaneously, enhancing both the speed and coverage of the search while also reducing the risk of prematurely discarding potentially useful paths due to early local estimation errors. To implement this, we simply delay the evaluation of states until after expanding the children of their children. In other words, instead of immediately evaluating each successor state, we first expand their successors, effectively going two levels deeper in the

search tree before performing any evaluations (Algorithm 2). Once this broader set of states is collected, we evaluate them all together in a single, parallel GNN forward pass.

Greedy Search (General Policy) In the literature of learning for planning, a heuristic function h defines a **general policy** if it can guide the agent to the goal greedily, without the need for backtracking or systematic search structures. As noted in Ståhlberg et al. [45], the heuristic does not have to be perfect.

Since several learning-based heuristics can be utilized as general policies, we also included a specific implementation designed to exploit this capability. We refer to this approach simply as *Greedy Search*.

Unlike traditional search algorithms, which maintain a priority queue and the $gMap$ to manage the search frontier and the exploration, this algorithm operates as a pure greedy execution loop. At each step, it generates the successors of the current state, evaluates them using the heuristic (or the general policy, since it is technically the same in this case), and selects the best one as the next state, discarding the rest.

This algorithm serves as a more interesting test for the quality of the learned model: it verifies whether the learned heuristic can actually function as a robust policy capable of solving the problem without relying on the safety net offered by the backtracking mechanisms used in standard search algorithms.

4.3 Adaptive Width Best-First Search

While the Multiple Evaluation (MBFS) and Deferred Multiple Evaluation (DBFS) strategies discussed in Section 4.2.3 focus on maximizing throughput by evaluating batches of states, they still operate with a static search logic. Standard Greedy

Best-First Search (GBFS) is structurally prone to getting stuck in search plateaus, regions where the heuristic function $h(s)$ offers no gradient information, or is misleading. In numeric planning, these plateaus are frequent due to the dense nature of the state space.

To address this, we introduce *Adaptive Width Best-First Search (AWBFS)*. Unlike standard GBFS, which evaluates the successors of a single state, or static approaches like *K-Best-First Search (KBFS)* [14], which expands the K best nodes at each iteration, AWBFS dynamically adjusts the "width" of the search frontier based on the heuristic's feedback. The core intuition is to maintain a narrow search width during promising descents to maximize speed, and broaden the search frontier only when the search stagnates, effectively using parallelism to escape search plateaus. We use the term width as used in the context of Beam Search literature, referring to the number of nodes expanded simultaneously.

4.3.1 Dynamic Width Adaptation

The behaviour of the algorithm is governed by two phases, determined by the heuristic value of the expanded states.

Greedy Phase (Descent) At the start of the search, or whenever a new state s is found, such that $h(s) < besth$ (where $besth$ is the best heuristic value observed so far), the improvement in the heuristic value means that the current search trajectory is promising. Consequently, the algorithm prioritizes exploitation over exploration. In this phase, *width* is kept steady, mimicking the behaviour of standard GBFS when the *width* is at 1.

Exploration Phase (Expansion) If the heuristic value of the expanded nodes stops improving (i.e. $h(s) \geq besth$), the algorithm assumes that the search has

encountered a plateau. To avoid getting stuck, the algorithm increases the *width*. This allows the search to broaden its frontier, processing multiple candidate paths in parallel, in order to find an exit from the uninformative region.

4.3.2 Memory Constraints with Learning-Based Heuristics

The implementation of AWBFS is particularly synergistic with learning-based heuristics that leverage GPU acceleration. Since heuristic evaluation is performed via optimized batch processing, increasing the batch size incurs minimal impact on the time required per iteration.

AWBFS exploits this by processing a larger set of candidate states when needed, without significantly slowing down the search cycle. However, the batch size cannot be increased arbitrarily. It is necessary to strictly control the expansion to prevent excessive memory usage, ensuring that the search never exceeds the maximum available memory limit m. By balancing the computational throughput against the finite memory capacity, AWBFS ensures robust performance without risking resource exhaustion.

To correctly predict the amount of memory that the algorithm will use before increasing the *width*, we need to calculate two things: the average memory cost per evaluation, and the estimated branching factor. The branching factor is the medium number of applicable actions in a set of states (i.e. the medium number of successors for each state). It can be formally defined as:

$$b = \frac{1}{|S|} \sum_{s \in S} |A(s)| \tag{4.5}$$

where

- b is the **branching factor**,
- S is the **set of considered states**,

- $A(s)$ is the **set of applicable actions** in state s,
- $|\cdot|$ denotes the **cardinality** of a set.

Calculating the branching factor is not an easy task, as it is not a static property of the domain but varies significantly across different regions of the state space. To address this, AWBFS relies on an online estimation mechanism. Instead of using a pre-determined fixed value, the algorithm calculates the average branching factor based on the ratio of actual successors generated (i.e., the number of evaluated states) to the number of expanded nodes. This dynamic estimates allows the system to predict the memory usage of future evaluations, adapting the *width* constraint to the specific density of the task to be solved.

Then, before incrementing the *width*, the algorithm checks if increasing the batch size would exceed the memory limit m. The estimated memory usage is calculated as $mem \cdot (bf \cdot (width+1))$, where mem represents the average memory cost per node, and bf is the estimated branching factor. This predictive check ensures that the expansion of the search frontier happens only if the predicted memory consumption remains within the available GPU memory.

4.3.3 Architectural Considerations

While AWBFS works particularly well for GPU architectures, it is important to explain the difference when used with standard CPU-based execution models. In particular, the trade-off between search width and computational time manifests differently across sequential, multi-core, and distributed settings.

Sequential Execution In a standard single-core implementation, evaluating a batch of N states requires N sequential calls to the heuristic function. Consequently, any increase in the search width results in a strictly linear increase in

the evaluation time. In this context, the dynamic widening strategy of AWBFS would become computationally expensive, as the ”cost” of escaping a plateau would directly delay the exploration of subsequent nodes.

Multi-core Parallelism Multi-threaded algorithms on a single machine can evaluate states in parallel, but they are limited by the number of physical cores. Furthermore, they often suffer from lock contention when accessing shared data structures like the Open List. Increasing the search width increases the frequency of these accesses, potentially degrading performance due to synchronization overhead.

Distributed Parallelism Distributed algorithms like HDA* (Hash Distributed A* [34]) partition the state space across multiple machines to overcome memory and compute limits. While this scales well, the bottleneck shifts to the communication overhead. Increasing the width implies generating and distributing a larger volume of states across the network, which can saturate the interconnect bandwidth and increase latency.

GPU-based Parallelism In contrast, GPUs rely on massive SIMD (Single Instruction, Multiple Data) parallelism. AWBFS exploits this by processing thousands of states in a single ”lock-step” cycle within the device’s high-bandwidth memory, avoiding both the lock contention of multi-core CPUs and the communication latency of distributed clusters.

Ultimately, this distinction does not imply that AWBFS is incompatible with CPU architectures. On the contrary, the core algorithmic principle of dynamic widening remains valid regardless of the hardware. However, its effective application on CPUs requires different implementation strategies and tuning. Unlike

the GPU setting, where memory is the primary constraint, CPU-based deployments would necessitate stricter controls on the expansion rate to balance the benefits of exploration against the linear time cost of sequential evaluation or the synchronization overheads of parallel execution.

4.3.4 Algorithm

The complete procedure is outlined in Algorithm 3. The inputs are the initial state of the problem s_o, the set of actions A, the goal of the problem G, the heuristic function h, and the maximum available GPU memory m. Lines 1–4 handle the initialization of the algorithm; mem represents the GPU memory used to evaluate one state. Lines 6–8 are a goal satisfaction check. At each iteration, if the goal is not reached, the algorithm checks if the heuristic value of the state extracted from the Open List is lower than the heuristic values observed until that point in the search (lines 9–10). The first time that this does not happen, the algorithm calculates the average branching factor of the problem (line 12). Then, the algorithm calculates all the successors of the current state (line 15), and if the *width* is greater than 1, it iteratively extracts additional states from the Open List until the number of extracted states in the iteration matches the current *width*. For each of these additional states, the algorithm checks if the goal condition G is satisfied; if not, it generates the state's successors and appends them to the cumulative list of successors (lines 17–21). Subsequently, the algorithm iterates over the cumulative set of successors to handle duplicate detection and pruning. If the successor is not currently in the $gMap$, or if it is present but with a higher path cost (g-value), it is added to the $gMap$ (replacing the existing entry in the case of a better path). Conversely, if a better or equal path to that state has already been found, the successor is removed from the list of successors. Successors are added to the $gMap$ directly inside the loop. This is important because

Algorithm 3 Adaptive Width Best-First Search (AWBFS)

Input: $\langle s_0, A, G \rangle$ — initial state, list of grounded actions, and goal.

h — heuristic function. m — GPU memory limit.

Output: Solution plan from initial state to goal, or none.

$Open \leftarrow \{s_0\}$ ▷ priority queue ordered by h

$gMap[s_0] \leftarrow \{0\}$ ▷ Maps state to best known g-value

$besth = \infty, bf = 0, width = 1$

$mem = \text{CalculateMemoryUsage}(s_o, A)$

while $Open \neq \emptyset$ **do**

  $s \leftarrow \text{Pop}(Open)$ ▷ state with lowest h

  if $\text{IsGoal}(s, G)$ **then**

    return $\text{ExtractSolution}(s, gMap)$

  else if $s.h < besth$ **then**

    $besth = s.h$

  else

    if $width == 1$ **then** $bf = \text{EvaluatedStates} \,/\, \text{ExpandedNodes}$

    if $mem \cdot (bf \cdot (width + 1)) < m$ **then**

      $width = width + 1$

  $successors = \text{Successors}(s, A)$

  if $width > 1$ **then**

    for $i \leftarrow 0$ **to** $width - 1$ **do**

      $s \leftarrow \text{Pop}(Open)$

      if $\text{IsGoal}(s, G)$ **then**

        return $\text{ExtractSolution}(s, gMap)$

      $successors \leftarrow successors \cup \text{SUCCESSORS}(s, A)$

  for all $s' \in successors$ **do**

    if $s' \notin gMap \vee gMap[s] + c(a, s) < gMap[s']$ **then**

      $gMap[s'] \leftarrow \{(gMap[s] + c(a, s))\};$

    else

      $successors \leftarrow successors \setminus \{s'\}$

  $\text{ParallelEvaluate}(successors, h)$ ▷ single GPU call

  $Open \leftarrow Open \cup successors$

return NONE

the algorithm evaluates successors from multiple parent states simultaneously. Therefore, it is highly probable that the same state could be generated via different parents within the same iteration. Immediate insertion ensures that these intra-iteration duplicates are effectively identified and filtered out (lines 22–26). Finally, the heuristic values for all successors are computed in a single *ParallelEvaluate* call, before adding the successors to the Open List and proceeding with the next iteration of the algorithm (lines 27–28).

4.3.5 Case Study: The First-Order Farmland Domain

To evaluate the algorithm in a context involving functional dependencies and setup costs, we consider the `Fo-Farmland` domain. This domain is an extension of the standard `Farmland` benchmark, which models the allocation of manpower to various farms to maximize agricultural output.

Problem Definition The core objective is to maximize the total benefit derived from a set of farms. The contribution of each farm to the total benefit is a function of the number of workers $x(f)$ assigned to it, weighted by a specific productivity factor w_f. The goal is satisfied when the weighted sum of manpower across all farms exceeds a specific threshold T:

$$\sum_{f \in Farms} w_f \cdot x(f) \geq T \tag{4.6}$$

For example, in a typical instance, the goal might be defined as $1.0 \cdot x(f_0) + 1.7 \cdot x(f_1) + 1.3 \cdot x(f_2) \geq 280.0$. To solve the problem, the planner must redistribute manpower from low-productivity farms (low w_f) to high-productivity ones (high w_f).

Standard vs. First-Order Dynamics The complexity of the domain lies in the trade-offs offered by the available transit actions:

- **Standard Domain:** The agent can choose between move-slow, which moves 1 unit of manpower losslessly, and move-fast. The move-fast action allows for higher throughput (moving 4 units at once) but is *lossy*: only 2 units reach the destination, representing a 50% resource loss. This creates a "greedy trap" where a heuristic might prefer the speed of move-fast without accounting for the long-term depletion of the total workforce.
- **First-Order Extension:** The `Fo-Farmland` variant replaces the lossy 'move-fast' action with a mechanism based on vehicle management. This introduces two coupled actions:
  1. `hire-car`: A setup action that increments the available fleet size ('num-of-cars'). This action consumes a step but produces no immediate movement of workers.
  2. `move-by-car`: A functional action that moves a variable amount of manpower, calculated as $4 \times$ num-of-cars. Unlike the standard 'move-fast', this action preserves manpower (workers are not lost), but it incurs a numeric cost proportional to the fleet size and the number of workers moved, which gets added to the threshold T in the goal.

Heuristic Ambiguity and Strategic Plateaus The introduction of the `hire-car` action creates a complex decision space where the optimal strategy is not fixed but highly instance-dependent. Depending on the specific resource thresholds and distances, the most efficient plan might range from a "pedestrian" strategy

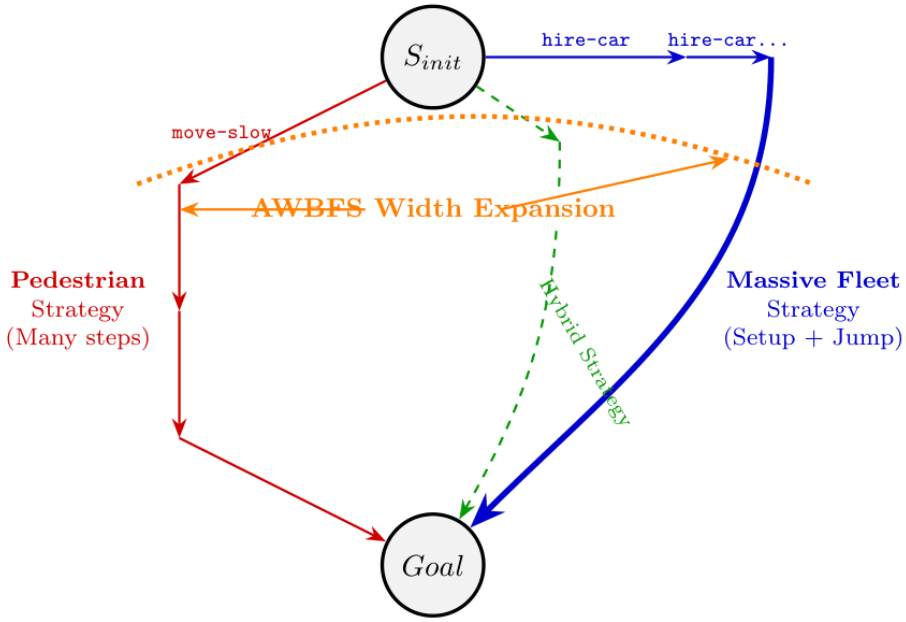


Figure 4.5: Topological overview of the search strategies in `Fo-Farmland`. From the initial state S_{init}, the planner can choose the **Pedestrian** route (left), which involves many small steps (`move-slow`); the **Massive Fleet** route (right), requiring an initial setup investment (`hire-car`) followed by a rapid transition; or a **Hybrid** route. The orange line illustrates how AWBFS expands the search width to capture these divergent strategies simultaneously.

(exclusively using `move-slow` to avoid setup costs), to a "massive fleet" approach (accumulating a large `num-of-cars` to move all workers in one go), or a hybrid "light transport" strategy (using a small fleet for iterative trips).

Heuristics often struggle to arbitrate between these competing global strategies (see Figure 4.5). Standard relaxation-based heuristics typically commit prematurely to the "pedestrian" strategy, as it offers immediate heuristic improvement, unaware of the long-term cumulative cost. Conversely, our GNN-based

heuristic might overfit to a ”massive fleet” strategy, investing heavily in setup costs even when the instance requires only minor adjustments. Crucially, neither strategy is universally superior; the optimality is entirely determined by the specific constraints of the instance.

AWBFS addresses this ambiguity through its dynamic width. When the strategy favored by the heuristic (e.g., the locally greedy `move-slow`) stagnates, the algorithm widens the frontier. This expansion forces the evaluation of ”sibling” strategies, effectively allowing the planner to explore the `hire-car` branches (and thus different fleet sizes) in parallel with the pedestrian approach. By not committing to a single heuristic gradient, AWBFS identifies the correct strategic investment for the specific instance without requiring a pre-defined policy.

4.3.6 Case Study: The Expedition Domain

To illustrate the algorithm's capability in handling strict resource constraints and necessary backtracking, we consider the `Expedition` domain. This domain models a logistic problem where a vehicle must traverse a distance larger than its supply capacity.

Domain Dynamics The problem involves sleds navigating a linear sequence of waypoints ($w_0 \rightarrow w_n$).

- **Resources:** Movement is governed by supplies. Each move_forwards or move_backwards action consumes 1 unit of supply.
- **Constraints:** Sleds have a hard sled_capacity. If the distance to the next supply depot is greater than the capacity, a single trip is impossible.
- **Storage:** Sleds can store_supplies at intermediate waypoints and retrieve_supplies later. This allows for ”staging”: moving supplies to an intermediate point,

returning to the source to restock, and then using the cached supplies to push further.

The instances of `Expedition` presents a deceptive topology for standard heuristics, from one particular instance of the domain:

- **Topology:** A start node wa_0 (rich in supplies), a goal node wa_5 that the sled has to reach, and a sequence of intermediate nodes ($wa_1 \dots wa_4$) with 0 supplies.
- **The Gap:** The distance from start to goal is 5 steps.
- **The Constraint:** The sled has a maximum capacity of 4 units.

Mathematically, $Distance(s_0, G) > Capacity(sled)$. A direct traversal is impossible. The sled will inevitably run out of supplies at wa_4, one step short of the goal.

Heuristic Failure and Dead-Ends Standard relaxation-based heuristics (such as h^{add} or h^{mrp}) are particularly susceptible to this topology. Because moving forward strictly reduces the estimated distance to the goal, these heuristics perceive the path $wa_0 \to wa_1 \to \dots \to wa_4$ as the optimal gradient. They fail to account for the "hard" resource constraint until the supplies are actually exhausted. Consequently, the search greedily drives the sled to wa_4, reaching a dead-end state where $supplies = 0$ and no further progress is possible. Crucially, any action to rectify this (e.g., move_backwards to refuel) initially increases the heuristic value, causing standard greedy search to prune or indefinitely delay the necessary backtracking. Our learned GNN-based heuristic is designed to be more sensitive to these resource traps, often successfully identifying that a direct path leads to failure. However, the `Expedition` domain is characterized by an extremely

high density of dead-ends. In harder instances, especially those involving multiple sleds or longer distances, the exact sequence of "staging" actions (dropping supplies, retreating, refilling, and retrieving) becomes computationally subtle. Even a sophisticated learned heuristic may occasionally guide the planner into a dead-end or fail to distinguish between a necessary retrogression and a useless loop.

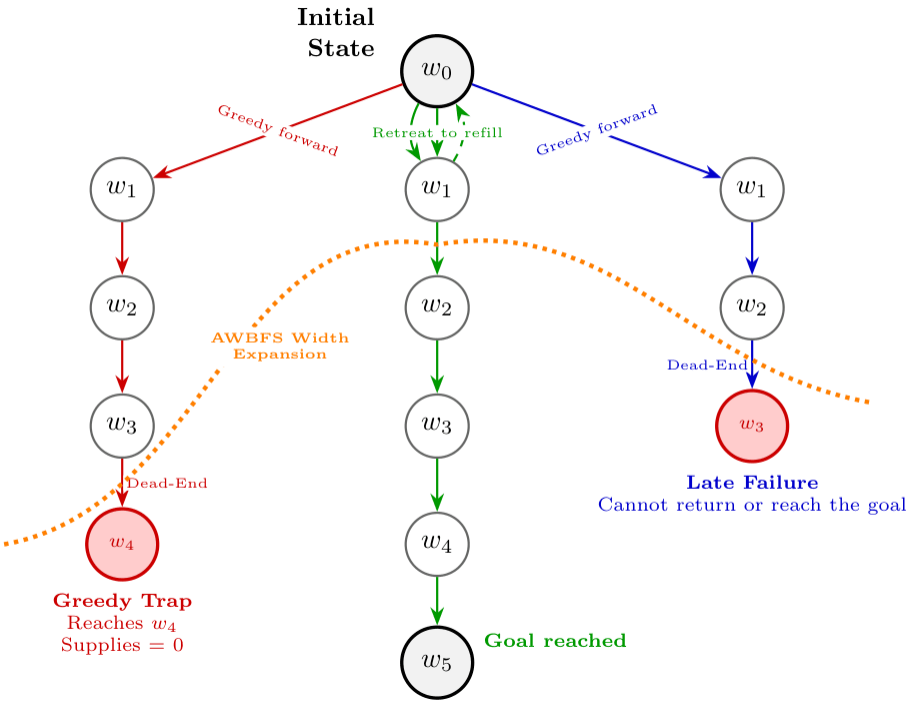


Figure 4.6: Detailed topology of the `Expedition` instance. The first path (left) exhausts supplies at w_4. The third path (right) gets stranded at w_3. The **Optimal** path (center) performs a retreat and refill maneuver ($w_0 \leftrightarrow w_1$) to cache supplies, enabling the final traversal to w_5.

AWBFS Resolution In these scenarios, AWBFS acts as a critical failsafe (see Figure 4.6).

1. **Stagnation Detection:** When the heuristic (standard or GNN) leads the search into a dead-end (e.g., reaching wa_4 empty) or into a region where $h(s)$ stops improving, AWBFS detects the plateau.

2. **Width Expansion:** The algorithm increases the search width, effectively looking beyond the single "best" successor.

3. **Recovering the Retreat:** By expanding multiple nodes in parallel, AWBFS is able to process the move_backwards and store_supplies actions even if they appear locally suboptimal. It discovers that the sequence of "retreat to refill" ($wa_1 \rightarrow wa_0$) enables a future state where the sled returns to wa_1 with full capacity plus the stored supplies, breaking the resource barrier.

Chapter 5

Experimental Analysis

This chapter presents a comprehensive experimental evaluation of the proposed framework. The primary motivation behind this analysis is to demonstrate that explicitly modeling the relational and logical structure of numeric conditions via Graph Neural Networks (GNNs) yields highly informative heuristics for numeric planning. Furthermore, since GNN evaluations naturally introduce a significant computational overhead compared to traditional methods, this chapter investigates how custom, highly parallelized search algorithms can effectively mitigate this cost and leverage the learned guidance. Specifically, the experimental analysis is designed to answer the following core research questions:

- **Informativeness:** Does explicitly incorporating numeric values and conditions into the GNN architecture improve heuristic guidance compared to both state-of-the-art traditional and learning-based approaches?
- **Computational Trade-offs:** How do different batch-aware search algorithms (such as MBFS, DBFS, and MA*) balance the trade-off between the computational cost of heuristic evaluation, search space exploration, and plan quality?

- **Heuristic Accuracy:** When stripping away the backtracking mechanisms of systematic search, how accurate is the raw guidance provided by the learned heuristic acting as a pure policy?
- **Adaptive Width Efficiency:** Does the newly introduced Adaptive Width Best-First Search (AWBFS) algorithm provide a strict improvement in coverage, speed, and plan quality over standard parallelized greedy approaches?

To address these questions, the chapter is organized as follows:

- **Section 5.1** details the selected benchmarks and the training setup, outlining the methodology used to generate the datasets and train the models.
- **Section 5.2** addresses the first question by evaluating the performance of the proposed numeric-sensitive heuristics against state-of-the-art baselines, highlighting the benefits of explicitly modeling numeric conditions.
- **Section 5.3** tackles the second and third questions by analyzing the different search algorithms implemented within the framework. It examines their computational trade-offs, memory usage, and demonstrates how they can be leveraged to isolate the intrinsic qualities of the learned heuristics.
- **Section 5.4** addresses the last question by evaluating the performance of AWBFS, demonstrating its impact on the overall efficiency of the planning process.

5.1 Benchmarks and Training Setup

As benchmarks, we selected nine domains from the International Planning Competition (IPC) 2023 Numeric Track [48]: `Counters`, `Fo-Counters`, `Sailing`,

Domain	$\mathbf{F_n}$	$\mathbf{F_{nc}}$	$\mathbf{C_n}$	$\mathbf{G_n}$	$\mathbf{C_{lin}}$	**Dead-Ends**
`Counters`	1.00	1.00	1.00	Yes	Simple	No
`Fo-Counters`	1.00	1.00	1.00	Yes	Linear	No
`Sailing`	0.46	0.66	0.88	No	Simple	No
`Fo-Sailing`	0.75	0.81	0.93	No	Linear	No
`Mprime`	0.08	0.05	0.28	No	Simple	No
`Expedition`	0.45	0.08	0.55	No	Simple	Yes
`Hydropower`	0.06	0.01	0.50	Yes	Simple	Yes
`Farmland`	1.00	0.21	1.00	Yes	Simple	No
`Fo-Farmland`	1.00	0.21	1.00	Yes	Linear	No

Table 5.1: Degree of numericity across different domains. **Legend:** $\mathbf{F_n}$: Percentage of numeric fluents; $\mathbf{F_{nc}}$: Percentage of numeric fluents and numeric constants; $\mathbf{C_n}$: Percentage of numeric conditions (action preconditions and goals); $\mathbf{G_n}$: Presence of numeric goals; $\mathbf{C_{lin}}$: Simple or linear numeric conditions.

`Fo-Sailing`, `Mprime`, `Expedition`, `Hydropower`, `Farmland`, and `Fo-Farmland`.

To ensure a comprehensive evaluation, these domains were specifically chosen to cover a wide spectrum of numeric complexity A detailed breakdown of the degree of numericity for each selected domain is provided Table 5.1. This analysis is crucial for understanding the inherent structure of the benchmarks. We evaluate the domains based on several features: the percentage of numeric fluents, the percentage of numeric fluents and numeric constants, the prevalence of numeric conditions in action preconditions and goals, the presence of numeric goals, the

Domain	**# Objects**		
	Train	**Validation**	**Test**
`Counters`	[4–7] counters	8 counters	[4–40] counters
`Fo-Counters`	[2–4] counters	[5–6] counters	[2–20] counters
`Sailing`	[1–2] boats, [1–5] people	4 boats, 2 people 1 boat, 6 people	[1–4] boats, [1–10] people
`Fo-Sailing`	1 boat, [1–2] people	1 boat, 3 people	[1–5] boats, [1–4] people
`Mprime`	[4–7] foods, [2–7] pains, [1–3] pleasures	10 foods, 1 pleasure, 7 pains 12 foods, 5 pleasures, 4 pains	[4–22] foods, [2–44] pains, [1–16] pleasures
`Expedition`	[6–8] waypoints	9 waypoints	[6–15] waypoints
`Hydropower`	[1010–1150] power	[1160–1190] power	[1010–2050] power
`Farmland`	[2–4] farms	4 farms	[2–10] farms
`Fo-Farmland`	2 farms	2 farms	[2–10] farms

Table 5.2: The table details the range of objects per split (train, validation, test) across different domains.

mathematical complexity of the numeric conditions (simple or linear), and the existence of dead-ends. Based on these metrics, the benchmark suite includes domains with a predominantly classical planning structure and low numericity (`Mprime`), simple numeric problems in which traditional heuristics excel (`Sailing`, `Counters`, `Farmland`), their linear numeric variants that are hard for traditional heuristics (`Fo-Counters`, `Fo-Sailing`, `Fo-Farmland`), and simple numeric problems that are hard for traditional heuristics due to the

Domain	**# Samples**	
	Train	**Validation**
`Counters`	200,000	41,000
`Fo-Counters`	89,000	30,000
`Sailing`	75,000	10,000
`Fo-Sailing`	50,000	12,000
`Mprime`	100,000	20,000
`Expedition`	45,000	12,000
`Hydropower`	15,000	2400
`Farmland`	150,000	50,000
`Fo-Farmland`	80,000	20,000

Table 5.3: The table details the number of samples used for training and validation, for each considered domain.

presence of dead-ends (`Expedition`, `Hydropower`). Each domain consists of 20 instances, with increasing difficulty scaling from the first instance to the last.

The datasets to train our networks is generated using an optimal numeric planner obtained by running A^* [25] with h^{rmax} [42]. The implementation is part of the ENHSP planning system that can be found at `https://github.com/hstairs/jpddlplus`.

For each domain, training is carried out on small instances, obtained by taking the smallest instances in the benchmark suite and creating new ones by changing the initial values. More precisely, we perform a random walk starting from the initial state, and use the reached state as the new initial state. We then solve the problem and store every traversed state of the solution in our dataset,

together with the actual distance to the goal. Details regarding the number of objects used in training, validation, and testing can be found in Table 5.2. The networks are therefore trained over tuples (s, v) where s is a state for a planning problem, and v is the optimal distance to the goal. We collected N tuples (s, v) for each domain, up to 40,000 tuples for each instance. Details regarding the number of samples used for each domain can be found in Table 5.3. The models are trained using the procedure and hyperparameters detailed in Section 4.1.5.

For deriving h^{ccWLF}_{rank}, we used the training procedure as for the instructions provided by the authors in Chen and Thiébaux [9].

The training process yields a model (heuristic function) that can be applied to any instance of the domain used for training. Experiments are run on a single Intel Xeon Gold 6140M (2.30GHz) core with a 5 minutes timeout for search and 8GB of memory. For our GNN heuristics, we use a NVIDIA v100 32GB GPU; for h^{ccWLF}_{rank}we follow the authors' suggestion and we do not use it. We kept an 8GB memory limit for our models, including both GPU and CPU memory.

5.2 Heuristics

The first experimental analysis aims at evaluating the performance of the proposed approach to learning heuristics for numeric planning problems. A question that the analysis tries to answer is whether our formulation improves on the state of the art in learning-based heuristics. We compare the heuristic learned by our architecture with one in which the numeric structure of the problem is completely ignored, namely h^0, which corresponds to Ståhlberg et al. [47] formulation, and with h^{ccWLF}_{rank}, the Graph-based heuristic proposed by Chen and Thiébaux [9], which is the state of the art in learning-based heuristics for numeric planning problems. To assess the performance gains from incorporating

numeric information, we experimented two variants of our learned heuristic: h^c, which corresponds to our architecture without the encoding of numeric values on the edges, i.e, where numeric conditions are abstracted as Boolean conditions, and h^n, which is our more advanced configuration where numeric values are provided to the MLP associated with each numeric condition. To ensure a fair comparison and assess which heuristic offers a better balance between informativeness and computational cost, all heuristics run using greedy best-first-search. That is, the search is guided by evaluation function $f(n) = h(n)$ with $h(n)$ be either h^0, h^c, h^n, or h^{ccWLF}_{rank}. This way, the planner becomes extremely sensitive to the quality of the heuristic. This choice aligns with the search algorithm used in the other systems [9, 42], ensuring a consistent basis for comparison. Heuristics h^0, h^c, h^n are run on board of LeapNP.

5.2.1 Heuristics Performance and Coverage Analysis

Table 5.4 reports results on the problem coverage obtained by all the considered heuristics. From these results it emerges quite clearly that our architecture benefits a lot from making it sensitive to the numeric information in the problem. Indeed, h^n consistently outperforms the other variants in most domains, except for `Mprime`. This is a domain with few numeric conditions, and the h^n infrastructure turns out to be overhead. Interestingly, `Mprime` is also the domain where h^{ccWLF}_{rank}shines. h^{ccWLF}_{rank}seems to be much more efficient when the domains have a small numeric component. Indeed, making exception for the two sailing domains (`Sailing`, `Fo-Sailing`) and the aforementioned `Mprime`, $h^{ccWLF}_{rank}$is outperformed by h^n. In `Mprime`, the numeric values are used only to check if there is at least one item of that type (i.e. if we want to take the action "drink", there is a precondition that checks if the number of drinks is at least 1). To reach the goal, knowing the numeric values is not important; knowing when

Domain	h^{ccWLF}_{rank}	h^0	h^c	h^n
Counters	3	N/A	2	**10**
Fo-Counters	4	N/A	2	**8**
Sailing	**5**	0	0	2
Fo-Sailing	**8**	1	2	**8**
Mprime	**13**	7	9	8
Expedition	3	1	1	**6**
Hydropower	N/A	0	1	**9**
Farmland	7	0	0	**10**
Fo-Farmland	4	2	4	**15**
Total	47	11	21	**76**

Table 5.4: Coverage analysis among three variants of our architecture (h^0,h^c,h^n) and h^{ccWLF}_{rank}. N/A indicates that the planner/heuristic does not support the features required by the domain.

a numeric condition is true or false is enough. The additional knowledge of h^n , in this case, just slows down the heuristic without adding useful information for reaching the goal. Overall, the tested approaches seem very complementary.

To analyze the tradeoff between guidance and computational cost, in Table 5.5 we show the coverage, the average time, and the average number of expanded nodes for each domain, comparing our final architecture with the state-of-the-art traditional heuristic h^{mrp} [42], implemented in the ENHSP planning system. The results indicate that h^n outperforms h^{mrp} in terms of coverage in 5 out of 9 domains (*Fo-Counters*, *Fo-Sailing*, *Expedition*, *Hydropower*, *Fo-Farmland*). A key insight emerges from the **Nodes** column: h^n consistently expands orders of magnitude fewer nodes than h^{mrp} to find a solution (e.g., in *Fo-Sailing*, 69 nodes vs. over 600,000). This drastic reduction in the

Domain	h^{mrp}				h^{n}			
	C	**T**	**N**	**L**	**C**	**T**	**N**	**L**
Counters	**12**	**3.06**	6272.9	**85.1**	10	13.89	**218.0**	100.3
Fo-Counters	5	1.47	8214.8	93.4	**8**	**0.68**	**9.4**	**9.4**
Sailing	**20**	**0.99**	**175.5**	**174.5**	2	13.08	180.5	180.5
Fo-Sailing	1	17.72	611,480.0	307.0	**8**	**6.20**	**69.0**	**69.0**
Mprime	**12**	**1.04**	25.6	**7.9**	8	1.42	**16.8**	10.3
Expedition	3	12.71	213,120.3	298.7	**6**	**3.15**	**58.3**	**58.0**
Hydropower	1	**0.77**	3818.0	26.0	**9**	2.26	**16.0**	**16.0**
Farmland	**20**	**1.21**	**234.6**	**233.6**	10	19.98	950.4	250.9
Fo-Farmland	5	52.15	531,201.7	148.3	**15**	**1.84**	**56.3**	**56.3**

Table 5.5: Performance comparison between h^{mrp} and h^{n}. Columns show coverage (C), average time (T) in seconds, average expanded nodes (N), and average plan length (L) calculated on the intersection of instances solved by both systems.

search space demonstrates the superior informativeness of the learned heuristic. h^{mrp} still performs better in the domains that include only simple numeric functions and do not have dead-ends.

In Table 5.6 we show the same values comparing our heuristic with h^{ccWLF}_{rank}. To ensure a fair comparison, the averages for time, expanded nodes and plan length were computed only on the instances successfully solved by both our heuristic and the one being compared. For Sailing, a comparison with h^{ccWLF}_{rank} was not possible due to the absence of commonly solved instances. Additionally, the results show that the decomposition introduced to handle goals involving an increasing number of numeric fluents is effective. This is particularly evident in the

Domain	h^{ccWLF}_{rank}				h^{n}			
	C	**T**	**N**	**L**	**C**	**T**	**N**	**L**
Counters	3	10.13	35.67	12.0	**10**	**0.40**	**8.33**	**8.3**
Fo-Counters	4	19.57	103,199.0	54.0	**8**	**0.53**	**7.25**	**7.3**
Sailing	**5**	–	–	–	2	–	–	–
Fo-Sailing	**8**	**9.74**	**176.5**	**176.5**	**8**	33.17	269.6	269.6
Mprime	**13**	28.91	137.86	20.0	8	**1.49**	**17.57**	**10.1**
Expedition	3	15.69	198,475.7	2019.3	**6**	**3.15**	**58.3**	**58.0**
Hydropower	N/A	–	–	–	**9**	7.05	96.7	38.7
Farmland	7	51.05	713,633.28	**252.42**	**10**	**7.18**	**275.28**	264.0
Fo-Farmland	4	34.57	364,013.5	185.0	**15**	**1.06**	**35.75**	**35.75**

Table 5.6: Performance comparison between h^{ccWLF}_{rank} and h^{n}. Columns show coverage (C), average time (T) in seconds, average expanded nodes (N), and average plan length (L) calculated on the intersection of instances solved by both systems, when possible. "–" denotes data not available due to lack of coverage or empty intersection.

Fo-Farmland domain, where our heuristic significantly outperforms h^{mrp}, in terms of guidance (fewer nodes expanded and better average plan length) and planning efficiency (lower computation time).

5.2.2 Comparison with h^{ccWLF}_{rank}

There are two key differences among Chen and Thiébaux [9] approach and ours. The first is that in their work, instead of GNNs, they use graph kernels to capture the relational structure of planning domains, leading to heuristics that are faster compared to GNN-based heuristics. In our work, we focus on the informa-

tiveness of our heuristics, leading to heuristics that are slower, but usually more informative.

The second difference is that their graph structure considers objects, grounded numeric or boolean fluents, propositional goals, and numeric fluents in the goals as nodes, and there are edges between an object and a fluent, or a goal, if the object is instantiated in the fluent or in the goal. Numeric values are incorporated directly in the node features of numeric fluents. However, numeric conditions. i.e., the logical expressions that must be satisfied for an action to be applicable or for a goal to be achieved (such as $x_1 + x_2 \leq 10$) are not explicitly represented in their graph. Instead, the graph only includes the numeric fluents themselves, and in the case of numeric goals, a scalar value called goal error is used to indicate how far the current state is from satisfying that numeric goal condition. This means that the structure of numeric conditions is flattened into a numeric distance, and the logical dependencies among objects involved in a condition are not captured structurally in the graph. In our work, instead, the nodes in the graph represent only the objects, and each Boolean fluent, Boolean goal, and grounded numeric condition (either a numeric precondition or a numeric goal) is represented as an edge connecting the objects involved. This allows us to reason directly over conditions, rather than fluents in isolation, capturing more faithfully the relational and logical structure of the numeric planning problem. This is particularly important in domains where the numeric relations themselves (rather than just the values of individual fluents) are essential to understanding progress toward the goal.

This difference can be seen in the experimental results of Table 5.6: in `Sailing` and `Fo-Sailing`, in which the goals are boolean conditions, and the numeric structure of the problem (the coordinates of boats and people that needs to be saved) is already encapsulated in the values of the grounded numeric flu-

ents, their architecture works better. On the other hand, in the domains where the goals are represented by numeric conditions (`Counters`, `Fo-Counters` `Farmland` and `Fo-Farmland`) and in those where numeric preconditions are important to capture the structure of the problem (`Expedition`), our architecture works significantly better. Unfortunately, we do not have a comparison for `Hydropower` since h^{ccWLF}_{rank} does not support it. In detail, `Hydropower` is excluded because it features a numeric effect not supported by Numeric Fast Downward [2], the planner underlying h^{ccWLF}_{rank}.

5.2.3 Scaling and Generalization Analysis

In classical planning, the optimal plan length of an instance is typically strictly correlated with the number of objects involved. Because the discrete logical structure is relatively rigid, to significantly increase the difficulty and the required plan length, one generally has to add more objects to the problem. Consequently, evaluating generalization in classical domains usually boils down to a single dimension: testing the model on instances with a higher number of objects.

In numeric planning, however, we must differentiate between the capability to generalize over higher plan lengths and the capability to generalize over a higher number of objects. In these domains, these two dimensions are largely decoupled. A problem's optimal plan length can grow exponentially not by adding new objects, but simply by altering a numeric constraint, a distance, or a resource capacity limit.

To measure both capabilities independently, a numeric version of the `Gripper` domain is used. Details regarding the domain are in appendix A. The heuristic for this domain has been trained using instances ranging from 10 to 30 balls and from 2 to 4 rooms, while validation was performed on instances with 25 and 30

Balls	Expanded Nodes	Plan Length	Optimal Length	Δ (%)
10	30	30	30	+0.00%
20	92	60	60	+0.00%
30	129	96	90	+6.67%
40	198	128	120	+6.67%
50	276	164	150	+9.33%
60	325	192	180	+6.67%
70	361	224	210	+6.67%
80	411	260	240	+8.33%
90	473	290	270	+7.41%
100	532	324	300	+8.00%
150	835	496	450	+10.22%
200	1276	680	600	+13.33%

Table 5.7: Analysis of generalization over plan length in the numeric `Gripper` domain. The number of rooms is kept constant (2 rooms), while the number of balls increases well beyond the training distribution (maximum 30 balls seen during training). Δ (%) highlights the relative suboptimality of the found solution compared to the optimal length.

balls across 5 rooms.

By structuring the domain this way, we can isolate the two axes of generalization. On one hand, by holding the number of rooms constant and increasing the initial number of balls, we can evaluate the model's ability to generalize to longer optimal plan lengths, requiring more pick, move, and drop actions. On the other hand, by keeping the number of balls fixed and increasing the number of rooms, we can assess the model's robustness to larger graph structures and increased node counts, without drastically skewing the numeric targets.

Rooms	Nodes	Plan	Opt.	Δ (%)	Rooms	Nodes	Plan	Opt.	Δ (%)
2	92	60	60	+0.00%	12	87	64	64	+0.00%
3	60	60	60	+0.00%	13	81	67	65	+3.08%
4	61	61	61	+0.00%	14	98	68	65	+4.62%
5	62	62	62	+0.00%	15	150	77	66	+16.67%
6	73	64	62	+3.23%	16	175	77	66	+16.67%
7	87	68	62	+9.68%	17	236	90	67	+34.33%
8	109	65	62	+4.84%	18	152	80	68	+17.65%
9	111	69	63	+9.52%	19	169	86	69	+24.64%
10	109	65	63	+3.17%	20	291	114	70	+62.86%
11	185	82	64	+28.13%	21	200	99	70	+41.43%

Table 5.8: Analysis of generalization over graph size (number of objects) in the numeric `Gripper` domain. The initial number of balls is kept constant (20 balls), while the number of rooms increases beyond the training distribution (maximum 4 rooms seen during training). Δ (%) highlights the relative suboptimality of the found solution compared to the optimal length.

The results of this bipartite analysis reveal distinct and highly informative behaviors regarding how the GNN-based heuristic degrades along the two different axes.

Table 5.7 illustrates the model's generalization capabilities over increasingly long plan lengths. In this scenario, the graph topology is kept strictly identical to the training distribution (2 rooms), while the continuous numeric variable (the number of balls) is scaled up to 200, far beyond the maximum of 30 balls seen during training. Remarkably, the heuristic demonstrates a graceful degradation. Even as the optimal plan length increases twentyfold, the relative error remains

tightly bounded.

Conversely, Table 5.8 paints a very different picture when scaling the graph topology. Here, the numeric difficulty is bounded (20 balls), while the number of discrete objects (rooms) increases well beyond the training distribution (trained on a maximum of 4 rooms). While the heuristic perfectly solves instances up to 5 rooms and maintains a low error rate up to 10 rooms, its performance becomes highly volatile and degrades sharply as the graph continues to grow. Past 15 rooms, the relative suboptimality spikes dramatically, reaching errors of over 40% and peaking at 62.86% for 20 rooms. Furthermore, the number of expanded nodes fluctuates unpredictably.

However, it is important to note that numeric scaling, even while maintaining a fixed topology, does not guarantee infinite graceful degradation. To further investigate the limits of generalizability over deep numeric horizons, a supplementary test was conducted on the `Sailing` domain. In this experiment, we kept only one person to save and one boat, while the numeric difficulty was isolated by progressively increasing the coordinate distance between the person and the boat. The results revealed a distinct threshold effect. The trained model exhibited remarkable precision up to a distance of 900, solving all instances optimally (which corresponds to an optimal plan length of 601 steps for the last one). Yet, when the distance was increased to 1000, requiring an optimal plan length of 669 steps, the heuristic experienced a sudden performance collapse, failing completely to find a solution. This total failure persisted for all subsequent distances.

This sharp drop-off in performance provides a clear and empirical explanation for the limited coverage observed earlier in the standard benchmark suite, where the model successfully solved only the first two instances of the Sailing domain. Because the third instance of the benchmark already requires an optimal plan length of 730 steps, it falls squarely beyond the model's critical generaliza-

Domain	$GBFS$			$MBFS$		
	Coverage	Time	Nodes/s	Coverage	Time	Nodes/s
Counters	8	41.7	22.24	**10**	**2.53**	**369.13**
Fo-Counters	6	8	18.6	**8**	**0.86**	**172.82**
Sailing	1	146.2	17.23	**2**	**10.52**	**239.63**
Fo-Sailing	**8**	70.4	66.7	**8**	**33.17**	**209.59**
Mprime	6	15	21.4	**8**	**0.62**	**521.83**
Expedition	**6**	45	18	**6**	**9.39**	**81.63**
Hydropower	8	8.2	18.49	**9**	**3.72**	**33.03**
Farmland	8	42.39	35.53	**10**	**7.90**	**176.67**
Fo-Farmland	12	65.87	34.45	**15**	**5.24**	**427.35**

Table 5.9: Comparison between $GBFS$ and $MBFS$ in terms of coverage, average time (seconds), and evaluated states per second. Average time and evaluated states per second are evaluated only in the instances solved by both systems.

tion threshold. At this depth, the heuristic gradient effectively vanishes, leaving the systematic search without meaningful guidance and unable to overcome the extreme length of the required trajectory.

5.3 Multiple Evaluation Algorithms Trade-offs

5.3.1 MBFS and DBFS

This experimental analysis aims at evaluating the performance of the framework with different search algorithms in the context of GNN-based heuristics. We compare the proposed variants, Multiple Evaluations and Deferred Multiple

Domain	*MBFS*				*DBFS*			
	C	**T**	**E**	**L**	**C**	**E**	**N**	**L**
Counters	**10**	**13.89**	464.5	100.3	**10**	28.96	**708.3**	**89.4**
Fo-Counters	**8**	32.98	170.3	22.0	7	**1.66**	**717.5**	**15.1**
Sailing	**2**	**13.08**	**290.1**	**180.5**	**2**	14.59	262.6	182.5
Fo-Sailing	**8**	**33.17**	209.6	269.6	**8**	70.47	**667.2**	**231.5**
Mprime	**8**	**1.42**	630.2	**10.2**	**8**	34.07	**812.8**	12.9
Expedition	6	**9.40**	81.6	**103.7**	**8**	11.49	**188.1**	165.3
Hydropower	**9**	**6.90**	29.3	**38.6**	**9**	8.11	**55.0**	38.8
Farmland	10	19.98	241.2	250.9	**13**	**7.23**	**786.2**	**246.0**
Fo-Farmland	**15**	5.74	419.3	174.5	13	**2.31**	**1409.3**	**51.3**

Table 5.10: Comparison between $MBFS$ and $DBFS$ in terms of coverage (C), average time (T) in seconds, evaluated states per second (E), and plan length (L). Average time, evaluated states per second, and plan length are evaluated only in the instances solved by both systems.

Evaluations, using GBFS as the baseline. To ensure a consistent basis for comparison, all search algorithms are evaluated using the same GNN-based heuristic and are integrated into the Python framework that we described before.

First, we examine the effectiveness of our multiple evaluation variant, which we called MBFS, against GBFS. From the results in Table 5.9, it emerges quite clearly that MBFS, when used with our GNN-based heuristic, is much faster than GBFS.

Then we evaluated MBFS against the deferred multiple evaluation variant, which we called DBFS. From the results in Table 5.10, DBFS consistently evaluates many more nodes per second compared to MBFS, this does not necessarily

Domain	GBFS	MBFS	DBFS
Expedition	659	677	765
Hydropower	667	685	767
Farmland	659	662	699
Fo-Farmland	657	659	811
Sailing	667	989	5015
Fo-Sailing	661	871	5959
Counters	663	1265	23,200
Fo-Counters	661	1265	24,687
Mprime	665	4861	>32,000

Table 5.11: GPU memory usage (in MB) comparison of *GBFS*, *MBFS* and *DBFS* for the most difficult instance of each domain.

translate into better performance. Although DBFS achieves slightly higher overall coverage and allows for the exploration of states that would otherwise be skipped, this comes at a cost: the additional evaluations lead to higher average runtimes in nearly all domains, without a clear improvement in plan quality. In other words, the ability to evaluate more nodes does not always correspond to more efficient or effective search behavior; additionally, DBFS is much more expensive in terms of GPU memory usage. In Table 5.11, we show the memory allocated for GBFS, MBFS, and DBFS for the most difficult instance of all the domains; we can identify three different groups of domains, based on their branching factor.

Therefore, we can identify the domains where even the hardest instances exhibit a low branching factor (Expedition, Hydropower, Farmland, Fo-Farmland), domains exhibiting an intermediate branching factor (Sailing

and `Fo-Sailing`), and domains with a high branching factor in the hardest instances (`Counters`, `Fo-Counters`, `Mprime`). For the first group, we can see that the differences in memory usage are minimal among the three variants; in those domains, the memory required to store the learned weights of the model is much higher than the memory required to store the graph representing the different states. In the second group, we observe that the memory usage for MBFS increases slightly, whereas for DBFS it already reaches values exceeding 5GB. Finally, in the third group, for `Counters` and `Fo-Counters`, memory consumption already increases with MBFS, although it remains within reasonable limits. In contrast, for `Mprime`, the domain characterized by the highest branching factor among those analyzed, memory usage approaches 5GB already with MBFS, and DBFS quickly saturates the available memory.

5.3.2 Optimization vs. Speed

While MBFS focuses on finding a solution as quickly as possible by aggressively following the heuristic gradient, many real-world applications require minimizing the cost of the plan. To support this, LeapNP also implements **MA*** (multiple evaluation A*), a batch-aware implementation of the A* algorithm. Like MBFS, MA* evaluates successors in parallel to maximize GPU throughput.

Table 5.12 compares the performance of MBFS against MA*. Interestingly, the results show that in the majority of the tested domains, MBFS produces plans with costs identical or very close to those found by MA*. This finding highlights the high informativeness of the learned heuristics, which can often find near-optimal paths, minimizing the need for extensive exploration. The only two notable exceptions are `Mprime` and `Fo-Farmland`, where MA* finds significantly better solutions. In terms of computational costs, MA* naturally pays a price in runtime and coverage compared to MBFS, as it requires exploring a

Domain	$MBFS$			MA*		
	Cov.	**Time**	**Plan Length**	**Cov.**	**Time**	**Plan Length**
Counters	**10**	**0.94**	**18.0**	5	44.43	**18.0**
Fo-Counters	**8**	**0.53**	**7.3**	4	3.22	**7.3**
Sailing	**2**	-	-	0	-	-
Fo-Sailing	**8**	**5.74**	67.5	2	117.01	**65.5**
Mprime	8	**1.42**	10.3	**9**	31.77	**6.8**
Expedition	**6**	**2.82**	**52.0**	4	7.73	**52.0**
Hydropower	**9**	**2.38**	35.0	5	20.29	**34.8**
Farmland	**10**	**10.18**	221.4	7	62.41	**211.4**
Fo-Farmland	**15**	**5.68**	177.1	11	29.76	**28.8**

Table 5.12: Comparison between $MBFS$ and MA* in terms of coverage, average time (seconds), and plan length. Average time and plan length are evaluated only in the instances solved by both systems.

larger portion of the state space.

5.3.3 Search vs. Policy

In this analysis, we investigate the intrinsic quality of the learned heuristics by stripping away the search mechanism. We compared MBFS against a pure Greedy Search. The Greedy Search approach treats the heuristic as a General Policy: at each step, it generates successors, evaluates them in a single batch, and deterministically selects the best one, discarding the rest.

Table 5.13 shows the comparison between MBFS and Greedy Search. The gap in the coverage between MBFS and Greedy Search reveals the imperfections of

Domain	$MBFS$			$GreedySearch$		
	Cov.	**Time**	**Plan Length**	**Cov.**	**Time**	**Plan Length**
Counters	**10**	**2.53**	**49.3**	8	14.47	280.8
Fo-Counters	**8**	32.98	22.0	7	**1.15**	**14.3**
Sailing	**2**	**13.08**	**180.5**	**2**	14.51	182.0
Fo-Sailing	**8**	33.17	269.6	**8**	**29.99**	**269.5**
Mprime	**8**	1.49	10.1	7	**0.65**	**6.6**
Expedition	**6**	2.82	**52.0**	4	**2.81**	**52.0**
Hydropower	**9**	2.78	**42.5**	2	**2.65**	**42.5**
Farmland	**10**	7.59	**302.7**	3	**7.26**	**302.7**
Fo-Farmland	**15**	**6.13**	**190.7**	9	7.31	214.9

Table 5.13: Comparison between $MBFS$ and $GreedySearch$ in terms of coverage, average time (seconds), and plan length. Average time and plan length are evaluated only in the instances solved by both systems.

the learned heuristics. We can observe that in Fo-Counters, Sailing and Fo-Sailing, both coverage and plan length remain strictly comparable. This indicates that the instances solved in these domains did not rely on the backtracking capabilities of MBFS, but depended almost exclusively on the accurate guidance provided by the learned heuristics. Conversely, on the other domains, the significant drop in coverage for Greedy Search highlights scenarios where the heuristics provide misleading guidance or lead to dead ends. In these cases, the systematic search structure of MBFS proves essential, acting as a safety net to recover from heuristic errors and escape local minima that a pure policy-based approach cannot overcome.

Domain	$MBFS$				$AWBFS$			
	C	**T**	**E**	**L**	**C**	**T**	**E**	**L**
Counters	10	**13.89**	464.5	100.3	**11**	17.72	**940.3**	**94.2**
Fo-Counters	**8**	29.30	171.3	23.8	**8**	**1.30**	**221.6**	**17.0**
Sailing	**2**	**13.08**	**290.1**	**180.5**	**2**	14.59	262.6	182.5
Fo-Sailing	**8**	33.17	209.6	269.6	**8**	**29.97**	**231.3**	**269.5**
Mprime	**8**	**1.42**	630.2	10.2	**8**	2.33	**690.7**	**9.8**
Expedition	6	9.40	81.6	103.7	**8**	**6.21**	**449.4**	**82.7**
Hydropower	**9**	7.05	29.1	**38.7**	**9**	**3.02**	**56.2**	**38.7**
Farmland	10	19.98	241.2	250.9	**14**	**7.45**	**467.0**	**237.6**
Fo-Farmland	15	25.14	401.9	274.6	**16**	**7.58**	**1117.5**	**132.1**

Table 5.14: Comparison between $MBFS$ and $AWBFS$ with the h^n heuristic, in terms of coverage (C), average time (T) in seconds, evaluated states per second (E), and plan length (L). Average time, evaluated states per second, and plan length are evaluated only in the instances solved by both systems.

5.4 Empirical Analysis of AWBFS

This final analysis investigates the efficacy of the proposed search algorithm AWBFS, against MBFS as the baseline. MBFS is selected as the primary term of comparison because it is structurally identical to the standard Greedy Best-First Search (GBFS), with the sole difference lying in the evaluation phase. Specifically, while maintaining the greedy expansion strategy, MBFS evaluates the successors of the node to be expanded in parallel. This modification is designed exclusively to minimize the computational time required by the GNN-based heuristic by leveraging batch processing, without altering the search trajectory compared

to a standard GBFS.

Table 5.14 reports results in terms of problem coverage, average time, average number of nodes evaluated per second, and average plan length. We compare AWBFS against a modified version of GBFS optimized for parallel evaluation. This variant retains the exact structural properties and search behavior of the standard algorithm but is optimized to evaluate all successors of the current state in parallel. This adjustment allows the baseline to leverage the GPU's computational power effectively, ensuring that the comparison focuses on the algorithmic efficiency of the search strategy rather than implementation bottlenecks. To ensure a fair comparison, the averages for time, expanded nodes per second, and plan length are computed only on the instances successfully solved by both algorithms.

The results show that AWBFS in this case performs way better than MBFS. In terms of coverage, it achieves superior results in 4 out of the 9 domains, while maintaining the same coverage in the remaining ones. This demonstrates that the adaptive width mechanisms provide a strict improvement over the baseline: it effectively extends the planner's capabilities in harder instances without degrading performance in domains where a simple greedy search is already sufficient. Additionally, the empirical data reveals that AWBFS outperforms the baseline in terms of efficiency and solution quality. Specifically, AWBFS achieves faster runtimes in 6 out of the 9 domains, and higher-quality solutions in 8 out of the 9 domains. The results in `Fo-Farmland` are particularly interesting. In this domain, although the coverage improves by only a single instance, AWBFS yields solutions of significantly higher quality, reducing the average plan length by more than half (132.1 vs 274.6). Remarkably, it achieves this improvement while reducing the computational time to approximately one-third of that required by the baseline (7.58s vs 25.14s).

Domain	$MBFS$	$AWBFS$	KBFS10	KBFS50	KBFS100
Counters	10	**11**	**11**	9	8
Fo-Counters	8	8	7	**9**	**9**
Sailing	**2**	**2**	**2**	**2**	1
Fo-Sailing	**8**	**8**	**8**	5	3
Mprime	**8**	**8**	**8**	**8**	**8**
Expedition	6	**8**	7	7	**8**
Hydropower	9	9	**10**	**10**	**10**
Farmland	10	14	14	**15**	**15**
Fo-Farmland	15	16	14	**17**	**17**
Total	76	**84**	81	82	79

Table 5.15: Coverage analysis among $MBFS$, $AWBFS$, and KBFS with the k-parameter set to 10, 50, and 100.

5.4.1 Comparison with KBFS

In this subsection, we extend our evaluation by comparing AWBFS against K-Best-First Search (KBFS)[14]. KBFS uses a width parameter K that selects the K most promising nodes from the open list and expands them together in a single iteration. To keep the comparison fair, we adapted KBFS to evaluate successor states altogether, making use of the GPU parallelization that we employ for the other search algorithms. Additionally, while we maintained an 8GB GPU memory limit for AWBFS, we did not impose any memory restriction on KBFS, allowing it to utilize the full 32GB available. We evaluate KBFS using three different configurations: K = 10, K = 50, and K = 100.

Table 5.15 summarizes the overall coverage across the different planners. AWBFS achieves the highest total coverage (84 solved instances), outperforming

KBFS by a small margin. Notably, despite having access to the full 32GB of GPU memory, the fixed batch size of KBFS proves to be highly memory-inefficient as the parameter K increases. Specifically, KBFS50 runs out of memory on several instances of the `Mprime` domain, and this issue is further exacerbated with KBFS100, which exhausts the available memory on multiple instances of both the `Mprime` and `Counters` domains. This demonstrates that dynamically adapting the search width is essential to prevent memory bottlenecks and out-of-memory errors.

To gain a deeper understanding of the search dynamics, Tables 5.16 and 5.17 provide a detailed comparison between AWBFS and the two best-performing KBFS variants (KBFS10 and KBFS50) across coverage, average time in seconds, evaluated states per seconds and average plan length.

Observing the tables, we can notice a pattern similar to what was previously discussed with DBFS: evaluating a significantly higher number of states per second does not necessarily translate to better overall performance.

Because KBFS uses a static K, even with $K = 10$ it evaluates on average more states than $AWBFS$, the only exception being the `Expedition` domain. Although the ability to immediately explore many different trajectories often yields better plan lengths, which is particularly evident in the case of KBFS50, AWBFS remains faster in the vast majority of domains. Ultimately, thanks to its dynamic adaptation mechanism, AWBFS avoids wasting resources on unpromising branches and manages to successfully solve a higher number of instances.

Domain	KBFS10				$AWBFS$			
	C	**T**	**E**	**L**	**C**	**T**	**E**	**L**
Counters	**11**	44.47	**923.3**	128.5	**11**	**31.40**	864.8	**120.2**
Fo-Counters	7	2.26	**736.7**	**14.1**	**8**	**1.07**	194.6	14.3
Sailing	**2**	34.96	**924.5**	**177.0**	**2**	**14.59**	262.6	182.5
Fo-Sailing	**8**	80.00	**736.2**	**249.9**	**8**	**29.97**	231.3	269.5
Mprime	**8**	**1.59**	**1184.9**	**6.8**	**8**	2.33	690.7	9.8
Expedition	7	**14.31**	534.0	138.0	**8**	27.84	**1211.6**	**130.3**
Hydropower	**10**	3.92	**188.6**	**38.4**	9	**3.02**	56.2	38.7
Farmland	**14**	12.07	**1017.4**	**266.4**	**14**	**8.43**	513.4	266.9
Fo-Farmland	14	**3.34**	**1539.5**	**38.1**	**16**	8.14	1184.4	133.6

Table 5.16: Comparison between KBFS10 and $AWBFS$ with the h^n heuristic, in terms of coverage (C), average time (T) in seconds, evaluated states per second (E), and plan length (L). Average time, evaluated states per second, and plan length are evaluated only in the instances solved by both systems.

Domain	KBFS50				$AWBFS$			
	C	**T**	**E**	**L**	**C**	**T**	**E**	**L**
Counters	9	45.45	**1284.8**	**63.3**	**11**	**12.26**	983.1	71.8
Fo-Counters	**9**	10.40	**986.8**	**16.8**	8	**1.30**	221.6	17.0
Sailing	**2**	123.87	**1151.4**	**175.0**	**2**	**14.59**	262.6	182.5
Fo-Sailing	5	74.04	**1129.5**	**119.2**	**8**	**16.09**	184.9	161.8
Mprime	**8**	12.14	577.2	**7.0**	**8**	**2.33**	**690.7**	9.8
Expedition	7	**21.51**	1093.9	**126.6**	**8**	27.84	**1211.6**	130.3
Hydropower	**10**	27.67	**118.4**	**38.2**	9	**3.02**	56.2	38.7
Farmland	**15**	24.11	**1644.1**	**261.1**	14	**8.43**	513.4	266.9
Fo-Farmland	**17**	11.68	**1814.3**	**34.6**	16	**7.74**	1210.1	121.4

Table 5.17: Comparison between KBFS50 and $AWBFS$ with the h^n heuristic, in terms of coverage (C), average time (T) in seconds, evaluated states per second (E), and plan length (L). Average time, evaluated states per second, and plan length are evaluated only in the instances solved by both systems.

Chapter 6

Conclusion and Future Work

In this thesis, we investigated the intersection of learning-based heuristics and numeric planning, with a specific focus on Graph Neural Network heuristics. The primary motivation behind this work was the observation that while numeric planning is essential for modeling real-world scenarios, it poses significant challenges for traditional heuristic search due to the infinite size of the state space and the complexity of numeric constraints. Our research aimed to bridge the gap between the expressive power of Graph Neural Networks and the challenges to use them for numeric planning problems.

6.1 Summary of Contributions

The contributions of this thesis can be summarized along three main axes:

A Novel GNN Architecture for Numeric Planning. We proposed a specialized GNN architecture that elevates grounded numeric conditions to first-class structural elements of the graph representation. Unlike previous approaches that treated numeric fluents merely as node features or flattened them into scalar val-

ues [9], our architecture explicitly encodes preconditions and goal conditions as edges. This design allows the message-passing mechanism to capture the rich relational dependencies defined by numeric constraints, enabling the network to learn more informative distance estimates. Experimental results on IPC benchmarks demonstrated that this approach significantly outperforms state-of-the-art learning-based baselines (like h^{ccWLF}_{rank}) and is competitive with traditional heuristics (like h^{mrp}) in domains characterized by complex structural relationships.

LeapNP: Bridging the Engineering Gap. Recognizing that the rigidity of existing C++ planning systems hinders the rapid prototyping required for deep learning research, we introduced LeapNP (Learning and Planning Framework for Numeric Problems). LeapNP is a modular, Python-based framework built upon Unified Planning, designed to lower the technical barrier for integrating neural networks into the planning loop. By providing a flexible interface for defining custom heuristics and search strategies, LeapNP facilitates experimentation with learning-based components without the overhead of low-level memory management.

GPU-Accelerated Search Algorithms. To address the computational bottleneck associated with neural inference, we developed novel search algorithms tailored for modern hardware. We introduced Multiple Evaluation Best-First Search (MBFS), which leverages batch processing to evaluate states in parallel, transforming the high latency of GNNs into an opportunity for higher throughput. Furthermore, we proposed Adaptive Width Best-First Search (AWBFS), a dynamic algorithm designed to mitigate the issue of heuristic plateaus. By expanding the search width only when the heuristic guidance stalls and strictly

monitoring GPU memory usage, AWBFS combines the speed of greedy search with the robustness of width-based exploration, proving effective in escaping local minima where standard GBFS fails.

6.2 Limitations

Despite the promising results, our approach is not without limitations. First, like all learning-based methods, the performance of our heuristic is heavily dependent on the distribution of the training data. While the lifted representation allows for generalization across problem sizes, the network may still struggle in instances that require structurally distinct reasoning patterns compared to those seen during training. Additionally, there are some limitations inherent to numeric planning problems:

- Numeric planning currently lacks a dedicated learning track, which means that publicly available training data is limited. Many numeric planning domains also lack instance generators, making it difficult to produce large and diverse datasets.

- Scaling remains a more significant obstacle in numeric planning. Finding valid plans often becomes computationally difficult even with a relatively small number of objects. As a result, the diversity in training instances, especially in terms of object counts, is typically low.

- In classical planning, suboptimal plans are sometimes used to enrich the training data, but generating useful suboptimal plans in numeric domains has proven problematic. Our experiments using suboptimal plans consistently led to large validation errors, and other recent work, such as Chen and Thiébaux [9], also relies exclusively on optimal plans.

- Finally, a distinctive difficulty in numeric planning is the high sensitivity of plan length to small changes in problem instances. For example, adding a single object can drastically increase the number of actions required to solve the problem. This poses a generalization challenge: a model trained only on plans with up to 100 actions may struggle to handle similar instances requiring 400 actions. This effect is evident in our experimental results, particularly in the `Sailing` domain, where even a small change in the problem leads to a significant increase in plan length and a corresponding drop in performance.

Regarding our framework, while LeapNP provides a highly modular and accessible environment for research, users should be aware of two primary limitations when applying it to practical problems:

- **Language-Specific Performance Overhead**: Since the framework is implemented entirely in Python, it prioritizes clarity and modifiability over raw execution speed. Consequently, when used with traditional, non-neural heuristics, LeapNP is inevitably slower in terms of nodes expanded per second compared to state-of-the-art planners written in low-level languages like C++ or Java. This framework is intended for rapid prototyping and research at the intersection of deep learning and planning, rather than for production environments requiring maximum search speed.

- **Memory Constraints in High-Branching Domains**: The parallel evaluation variants (MBFS, DBFS, and MA*) rely on batching multiple states into a single GPU forward pass. Our experiments show that in domains with a high branching factor, such as Mprime or Counters, the memory required to store and process these batches can grow significantly. For very large instances or hardware with limited GPU memory, this can lead

to memory saturation, acting as a physical constraint on the search depth and batch size.

With AWBFS, we explicitly aimed to address this latter constraint. By incorporating a dynamic memory monitoring mechanism, the algorithm ensures that the search process remains within the available hardware limits, automatically checking GPU usage before expanding new batches. This design choice effectively mitigates the risk of memory saturation, allowing the planner to operate robustly even in high-branching domains where standard parallel evaluation strategies might otherwise fail due to resource exhaustion.

6.3 Future Work

6.3.1 Numeric HER

A major bottleneck identified in this thesis is the reliance on existing planners to generate training data (optimal or valid plans), which limits scalability. A promising avenue to overcome this is to adapt Hindsight Experience Replay (HER) to the numeric planning setting. Recently, Ståhlberg and Geffner [44] demonstrated the effectiveness of *Propositional HER* and *Lifted HER*, where failed trajectories are relabeled as successful examples for the specific set of atoms actually achieved. Extending this concept to numeric state variables presents a novel challenge: unlike boolean atoms, numeric goals are often defined by inequalities over continuous ranges. Future work could investigate a Numeric HER mechanism that generates *post-hoc* numeric goals satisfied by the reached state (e.g., creating a goal $x \geq k$ where k is the value reached in the trajectory). This would allow the GNN to learn valid heuristic estimates from random or exploratory interactions with the environment, drastically reducing the dependence on supervision from

symbolic planners.

6.3.2 Learning Numeric Macro-Actions

As discussed in the limitations (Section 6.2), the performance of our approach tends to degrade in problems requiring very long plans, such as specific instances of the Sailing domain. This is a known challenge in numeric planning, where a goal state might require applying the same operator hundreds of times (e.g., incrementing a counter or moving by small steps). Future research should investigate the use of GNNs to identify and learn numeric macro-actions. Instead of strictly estimating the distance to the goal, the network could be trained to predict "jumps" in the state space, essentially parameterizing a sequence of repetitive actions into a single step. This would effectively reduce the search depth and mitigate the horizon effect that currently limits scalability in large numeric instances.

6.3.3 Multi-Queue Search Strategies

Our experimental analysis revealed a significant complementarity between our learned heuristic (h^n) and traditional relaxation-based heuristics like h^{mrp}. While h^n excels in domains that require handling linear numeric expressions and dead-ends, but h^{mrp}, as well as other traditional heuristics, work better in domains that only require handling simple numeric expressions. A promising direction is to integrate these diverse estimators within a Multi-Queue Best-First Search framework. By maintaining separate priority queues for the learned heuristic and a traditional numeric heuristic, and alternating between them during the expansion phase, the planner could leverage the structural guidance of the GNN while retaining the reliability of traditional methods, potentially improving both

coverage and robustness across heterogeneous domains.

Appendix A

The Numeric Gripper Domain

In the classical formulation of Gripper, a robot equipped with two grippers must transport a set of balls from one room to another. Crucially, in the classical version, every single room, gripper, and ball is represented as a distinct, discrete object. The numeric formulation used, instead, keeps the grippers and the rooms as discrete objects, while the number of balls is a numeric fluent assigned to each room object.

Under this modeling paradigm, the state representation relies on a hybrid structure. The static topology of the relational graph is defined purely by the discrete objects: the rooms and the grippers. The logical state is tracked via Boolean predicates that symbolize the robot's current location and whether a specific gripper is free or currently carrying a ball. The numeric state is via the numeric fluents tracking the amount of balls currently located in a given room.

The domain actions are adapted to manipulate these numeric fluents alongside the logical predicates. The pick action requires the robot to be in a specific room and a chosen gripper to be free, alongside a numeric precondition checking that the number of balls in the room is strictly greater than zero. Upon execution,

```
(define (domain gripper)

(:types
   room - object
   gripper - object
)

(:predicates ;
   (pos_robby ?r -room)
   (free ?g -gripper)
   )


(:functions
   (balls_num ?r -room)
)

   (:action move
       :parameters  (?from ?to - room)
       :precondition (and  (pos_robby ?from))
       :effect (and  (pos_robby ?to)
                     (not (pos_robby ?from))))

   (:action pick
       :parameters (?r -room ?gripper -gripper)
       :precondition  (and (>= (balls_num ?r) 1)
                           (pos_robby ?r) (free ?gripper))
       :effect (and
                    (not (free ?gripper))
                    (decrease (balls_num ?r) 1)))

   (:action drop
       :parameters   ( ?r -room ?gripper -gripper)
       :precondition  (and
                           (not (free ?gripper)) (pos_robby ?r))
       :effect (and
                    (free ?gripper)
                    (increase (balls_num ?r) 1)))
)
```

Figure A.1: Domain of the numeric gripper problem

it decreases the numeric fluent representing the balls in that room by one, and updates the gripper's state to indicate it is holding a ball. Conversely, the drop action updates the corresponding gripper to be free and increases the ball count of the current room. The move action simply updates the Boolean predicates denoting the robot's location.

Figure A.1 illustrates the PDDL encoding of the domain, while Figure A.2 shows an example of a problem instance.

```
(define (problem p_3_20) (:domain gripper)
(:objects
   g1 g2 - gripper
   r1 r2 r3 - room
)

(:init
   (= (balls_num r1) 20)
   (= (balls_num r2) 0)
   (= (balls_num r3) 0)
   (pos_robby r2)
   (free g1)
   (free g2)
)

(:goal (and
   (= (balls_num r2) 10)
   (= (balls_num r3) 10)
))

)
```

Figure A.2: An instance of the numeric gripper problem

Bibliography

[1] Ralph Abboud, İsmail İlkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. The surprising power of graph neural networks with random node initialization. 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 2112–2118. ijcai.org, 2021. doi: 10.24963/IJCAI.2021/291. URL https://doi.org/10.24963/ijcai.2021/291.

[2] Johannes Aldinger and Bernhard Nebel. Interval based relaxation heuristics for numeric planning with action costs. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 15–28. Springer, 2017.

[3] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. Pyperplan, 2020. URL https://doi.org/10.5281/zenodo.3700819.

[4] Maciej Besta and Torsten Hoefler. Parallel and distributed graph neural networks: An in-depth concurrency analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

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

[6] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[7] Valerio Borelli, Alfonso Emilio Gerevini, Enrico Scala, and Ivan Serina. Leapnp: A modular python framework for benchmarking learned heuristics in numeric planning. *Future Internet*, 18(2):93, 2026.

[8] Ethan Burns, Sofia Lemons, Wheeler Ruml, and Rong Zhou. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39:689–743, 2010.

[9] Dillon Chen and Sylvie Thiébaux. Graph learning for numeric planning. *Advances in Neural Information Processing Systems*, 37:91156–91183, 2024.

[10] Dillon Z Chen, Sylvie Thiébaux, and Felipe Trevizan. Learning domain-independent heuristics for grounded and lifted planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 20078–20086, 2024.

[11] Dillon Z Chen, Felipe Trevizan, and Sylvie Thiébaux. Return to tradition: Learning reliable heuristics with classical machine learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pages 68–76, 2024.

[12] Augusto B Corrêa, André G Pereira, and Jendrik Seipp. Classical planning with llm-generated heuristics: Challenging the state of the art with python code. *arXiv preprint arXiv:2503.18809*, 2025.

[13] Rüdiger Ebendt and Rolf Drechsler. Weighted a* search–unifying view and application. *Artificial Intelligence*, 173(14):1310–1342, 2009.

[14] Ariel Felner, Sarit Kraus, and Richard E Korf. Kbfs: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, 39(1):19–39, 2003.

[15] Patrick Ferber, Malte Helmert, and Jörg Hoffmann. Neural network heuristics for classical planning: A study of hyperparameter space. In *ECAI 2020*, pages 2346–2353. IOS Press, 2020.

[16] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[17] Maria Fox and Derek Long. Pddl+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, volume 4, page 34, 2002.

[18] Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003. doi: 10.1613/JAIR.1129. URL https://doi.org/10.1613/jair.1129.

[19] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60, 2017.

[20] Héctor Geffner and Nir Lipovetzky. Width and serialization of classical planning problems. 2012.

[21] Alfonso Gerevini, Ivan Serina, et al. Lpg: A planner based on local search for planning graphs with action costs. In *Aips*, volume 2, pages 281–290, 2002.

[22] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning with numerical expressions in lpg. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 667–671, 2004.

[23] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl — the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, October 1998.

[24] William L Hamilton. *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[25] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[26] Malte Helmert. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, pages 44–53, 2002.

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

[28] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3): 57–57, 2001.

[29] Jörg Hoffmann. The metric-ff planning system: Translating"ignoring delete lists"to numeric state variables. *Journal of artificial intelligence research*, 20: 291–341, 2003.

[30] Rostislav Horčík and Gustav Šír. Expressiveness of graph neural networks in planning domains. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pages 281–289, 2024.

[31] Rostislav Horčik, Gustav Šír, Vítězslav Šimek, and Tomáš Pevnỳ. State encodings for gnn-based lifted planners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26525–26533, 2025.

[32] Rushang Karia and Siddharth Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 8064–8073, 2021.

[33] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363, 1992.

[34] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195:222–248, 2013.

[35] Ryo Kuroiwa and Alex Fukunaga. Analyzing and avoiding pathological behavior in parallel best-first search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 175–183, 2020.

[36] Andrea Micheli, Arthur Bit-Monnot, Gabriele Röger, Enrico Scala, Alessandro Valentini, Luca Framba, Alberto Rovetta, Alessandro Trapasso, Luigi Bonassi, Alfonso Emilio Gerevini, Luca Iocchi, Felix Ingrand, Uwe Köckemann, Fabio Patrizi, Alessandro Saetti, Ivan Serina, and Sebastian Stock. Unified planning: Modeling, manipulating and solving ai planning problems in python. *SoftwareX*, 29: 102012, 2025. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2024.102012. URL https://www.sciencedirect.com/science/article/pii/S2352711024003820.

[37] Hootan Nakhost, M Müller, R Valenzano, and F Xie. Arvand: the art of random walks. *The*, pages 15–16, 2011.

[38] Wiktor Piotrowski, Alexandre Perez, and Sachin Grover. Nyx: Planning for emerging problems with pddl+ and beyond. 2024.

[39] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[40] Nicholas Rossetti, Massimiliano Tummolo, Alfonso Emilio Gerevini, Luca Putelli, Ivan Serina, Mattia Chiari, and Matteo Olivato. Learning general policies for planning through gpt models. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pages 500–508, 2024.

[41] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramirez. Interval-based relaxation for general numeric planning. In *ECAI 2016*, pages 655–663. IOS Press, 2016.

[42] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez. Subgoaling techniques for satisficing and optimal numeric planning. *J. Artif. Intell. Res.*, 68:691–752, 2020. doi: 10.1613/JAIR.1.11875. URL `https://doi.org/10.1613/jair.1.11875`.

[43] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 574–584, 2020.

[44] Simon Ståhlberg and Hector Geffner. First-order representation languages for goal-conditioned rl. *arXiv preprint arXiv:2512.19355*, 2025.

[45] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning generalized policies without supervision using gnns. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 19, pages 474–483, 2022.

[46] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 629–637, 2022.

[47] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Muninn. *Tenth International Planning Competition (IPC-10) Learning Track: Planner Abstracts*, 2023.

[48] Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fiser, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 international planning competition. *AI Mag.*, 45(2):280–296, 2024. doi: 10.1002/AAAI.12169. URL `https://doi.org/10.1002/aaai.12169`.

[49] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[50] Ryan Xiao Wang and Sylvie Thiébaux. Learning generalised policies for numeric planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pages 633–642, 2024.

[51] Fan Xie, Martin Müller, and Robert Holte. Adding local exploration to greedy best-first search in satisficing planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[52] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):249–270, 2020.