Learned Pairwise Rankings for Greedy Best-First Search

Mingyu Hao[1], Felipe Trevizan[1], Sylvie Thiébaux[1,2], Patrick Ferber[3,4], Jörg Hoffmann[4]

[1]Australian National University, Australia
[2]LAAS-CNRS, Université de Toulouse, France
[3]University of Basel, Switzerland
[4]Saarland University, Germany
{mingyu.hao,felipe.trevizan,sylvie.thiebaux}@anu.edu.au, patrick.ferber@unibas.ch, hoffmann@cs.uni-saarland.de

Abstract

We propose a new approach based on ranking to learn to guide Greedy Best-First Search (GBFS). As previous ranking approaches, ours is based on the observation that directly learning a heuristic function is overly restrictive, and that GBFS is capable of efficiently finding good plans for a much more flexible class of total quasi-orders over states. In order to learn an optimal ranking function, we introduce a new ranking framework capable of leveraging any neural network regression model and efficiently handling the training data through batching. Compared with previous ranking approaches for planning, ours does not require complex loss functions and allows training on states outside the optimal plan with minimal overhead. Our experiments on the domains of the latest planning competition learning track show that our approach substantially improves the coverage of the underlying neural network models without degrading plan quality.

1 Introduction

Greedy Best-First Search (GBFS) is one of the most widely used algorithms in satisfying planning. GBFS uses a heuristic function to estimate the cost to reach the goal from a given node. The nodes are put in a priority queue based on their heuristic values, and the node with the lowest heuristic value is expanded next. This strategy will return a plan if one exists, but may result in suboptimal solutions. Many researchers have made significant efforts to design informative heuristics to guide GBFS (Bonet, Loerincs, and Geffner 1997; Hoffmann and Nebel 2001; Richter and Westphal 2010), however the efficiency of the search depends on properties of the heuristic function which are not yet completely understood (Wilt and Ruml 2016; Cohen and Beck 2018).

Therefore, in recent years, the possibility of exploring the space of heuristic functions using machine learning has raised increasing attention (Arfaee, Zilles, and Holte 2011; Shen, Trevizan, and Thiébaux 2020; Ferber, Helmert, and Hoffmann 2020; Karia and Srivastava 2021; Ståhlberg, Bonet, and Geffner 2022; Heller et al. 2022; Chen, Thiébaux, and Trevizan 2024). These approaches try to learn or improve a heuristic function for a planning domain or, occasionally, a single large problem instance. They typically formulate the learning problem as one of regression, and train models using pre-computed state-value pairs extracted from (often optimal) plans generated by off-the-shelf planners for small problem instances.

However, labelling states with informative values and attempting to learn accurate values for the heuristic is expensive and unnecessary for the purpose of learning to guide GBFS. This is because the choice of node expansion by GBFS is not determined by the actual numerical values of the heuristic but by the relative ordering over nodes they induce (Röger and Helmert 2010). Therefore, regression models, which attempt to learn precise heuristic values, artificially restrict the target hypothesis space and the training dataset one can learn from. They also introduce issues such as the difficulty of distinguishing between states that have close heuristic values. In fact, for GBFS to find the optimal plan, it suffices for the heuristic to order the states on the optimal path as preferable to those off it. This is called the perfect ranking heuristic in (Chrestien et al. 2023) as it also minimises search effort.

These considerations motivate an alternative framing of the problem as one of learning to rank pairs of states, building on the active research on learning to rank in information retrieval and relevance analysis (Burges 2010; Damke and Hüllermeier 2021; Li 2022). Garrett, Kaelbling, and Lozano-Pérez (2016) were the first to approach this problem for planning by using a statistical machine learning model, namely RankSVM (Joachims 2002), hand-crafted domain-independent features, and a loss function measuring the normalised difference between the number of correctly and incorrectly classified pairs.

In contrast, this paper explores pairwise ranking with modern deep-learning methods. We propose a novel framework combining a Direct Ranker (Köppel et al. 2020b) with any existing neural network regression model for learning heuristics, which are trained end-to-end in order to learn a total quasi-order, i.e. a total, transitive, and reflexive relation over pairs of states. This framework does not require complex loss functions, and exploits the transitivity of the ordering relation to learn a ranking just slightly stronger than the perfect ranking, from an order of magnitude less data than alternative approaches. Moreover, it supports efficient batched evaluation to speed up training.

We evaluate our approach on the learning track of

the 2023 international planning competition, using the recent STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020), GOOSE-ILG (Chen, Trevizan, and Thiébaux 2024) and GOOSE-LLG (Chen, Thiébaux, and Trevizan 2024) domain-independent graph neural network architectures, as the underlying regression model. We show that our ranking approach considerably improves the coverage of both these models and expands their ability to generalise to larger problems than those trained on, whilst preserving their plan quality and time to convergence. We also compare our approach with the recently and independently developed work from Chrestien et al. (2023). We show that our framework requires less training data and generalises better to unseen problems.

2 Background

We start by introducing the necessary background on planning, greedy best-first search, and learning to rank. We use $[\![n]\!]$ to denote $\{1, 2, ..., n\}$.

2.1 Planning

We consider classical planning problems described by a tuple $\mathbb{P} = \langle \mathsf{S}, s_0, \mathsf{G}, \mathsf{A}, C \rangle$ where S is a finite and discrete set of states, s_0 is the initial state, $\mathsf{G} \subseteq \mathsf{S} \setminus \{s_0\}$ is the set of goal states, A is the set of actions of which A_s is the subset applicable in state s, and an action $a \in \mathsf{A}_s$ is a deterministic transition function, which takes state s as input and outputs the successor state s'. In the following, we write N_s for the set of successors of a state s: it consists of all states s' such that there exists an applicable action $a \in \mathsf{A}_s$ with $s' := a(s)$. The cost function $C(a)$ returns the non-negative cost of applying action a. A unit-cost problem is one for which all actions have cost 1. A plan for $\mathbb{P}$ is a sequence of actions $a_1, \ldots, a_n$ that induces a sequence of states (a path) $s_0, s_1, ..., s_n$ where $a_i \in \mathsf{A}_{s_{i-1}}, s_i := a_i(s_{i-1})$ for $i \in [\![n]\!]$, and $s_n \in \mathsf{G}$. A plan is optimal if its cost $\sum_{i \in [\![n]\!]} C(a_i)$ is minimal. In the unit-cost case, optimal plans are plans of minimal length.

Greedy Best-First Search (GBFS) is the most commonly used algorithm for satisficing planning. In order to guide the search towards the goal, GBFS relies on a heuristic function $h : \mathsf{S} \to \mathbb{R}$ returning an estimate $h(s)$ of the cost to reach the goal from a given state s, and always expands the state with the lowest heuristic value on the search frontier. As shown in Algorithm 1, GBFS maintains two lists of nodes[1]: the open list ($\mathcal{O}$) – a priority queue ordered by increasing h value, containing the nodes that have been generated but not yet expanded, and a closed list ($\mathcal{C}$) containing previously expanded nodes which is used to avoid revisiting the same state twice and creating loops. At each iteration, the node at the front of the open list – the node with the lowest h value – is transferred to the closed list and expanded by generating its successors. The search stops if a successor is a goal state. Otherwise, any unvisited successor is added to the open list.

GBFS is guaranteed to find a plan if one exists, but the plan may not be optimal. In fact, even when guided with the h^* heuristic which returns the optimal cost to reach the goal, GBFS is only guaranteed to return the optimal plan for unit cost problems. In the following, we assume that problems have unit cost.

[1]In the interest of readability, Algorithm 1 identifies nodes with states, whereas nodes additionally record the h value of the state, the parent node, and the action applied at the parent.

Algorithm 1: GBFS Algorithm

Input: problem $\mathbb{P}$, heuristic function h
Output: Plan π or *unsolvable*

```
1:  O := {s_0}
2:  C := ∅
3:  while O ≠ ∅ do
4:      s := argmin_{s'∈O} h(s')
5:      O := O \ {s}
6:      C := C ∪ {s}
7:      for all s' ∈ N_s do
8:          if s' ∈ G then
9:              return Extract-Solution(s')
10:         end if
11:         if s' ∉ C ∪ O then
12:             O := O ∪ {s'}
13:         end if
14:     end for
15: end while
16: return unsolvable
```

2.2 Learning to Rank

Learning-to-rank is a well-studied field in machine learning (Liu et al. 2009; Li 2022). Given a set of objects, in our case a set of states, $\mathsf{B} = \{s_1, \ldots, s_k\}$, it aims to learn an order $\preceq$ that results in the ranking $s_{i_1} \preceq s_{i_2} \preceq \ldots \preceq s_{i_k}$, where $i_1, \ldots, i_k$ is a permutation of $[\![k]\!]$. In this work, we further require the order to have the following properties:

Definition 1 (Total Quasi-Order $\preceq$)**.** *A total quasi-order $\preceq$ over a set X is a binary relation with the following properties*

- *Totality:* $\forall a, b \in X \;\; a \preceq b \vee b \preceq a$*;*
- *Transitivity:* $\forall a, b, c \in X \;\; a \preceq b \wedge b \preceq c \Rightarrow a \preceq c$*.*

Moreover, totality implies

- *Reflexivity:* $\forall a \in X \;\; a \preceq a$*.*

We define the strict order $\prec$ in the obvious way: $a \prec b$ iff $a \preceq b$ and $b \not\preceq a$. Note that transitivity also holds for the strict order $\prec$: $\forall a, b, c \in X \;\; a \prec b \wedge b \prec c \Rightarrow a \prec c$.

There are several different approaches to solving the learning-to-rank problem, such as SVM (Shashua and Levin 2002; Herbrich, Graepel, and Obermayer 1999), Boosting (Freund et al. 2003; Wu et al. 2010) and Neural Networks (Burges et al. 2005; Köppel et al. 2020b). Based on how the order is represented, learning-to-rank approaches are divided into three groups: *pointwise*, *pairwise* and *listwise* (Li 2022). Listwise approaches learn to sort a list of objects to match a desired total order. For planning, this approach is infeasible since we do not know a priori the list of states we need to sort.

The pointwise approach represents the total quasi-order by a ranking function $r : \mathsf{B} \to \mathbb{R}$ which takes in one object and returns a numerical score (Li 2022). This numerical

score is then used to sort B. Note that GBFS uses the heuristic h as a pointwise ranking function: the set of open states $\mathcal{O}$ is sorted according to their h values (Line 4 of Algorithm 1). Similarly, learning a heuristic function can be seen as learning a pointwise ranking function with extra constraints imposed by the definition of a heuristic function, e.g., $h(s) \geq 0$ for all s, $h(g) = 0$ for all $g \in \mathsf{G}$, or even admissibility or consistency.

Lastly, a pairwise ranking function takes a pair of elements as input and it outputs the ordering between the two objects. Formally, a pairwise ranking function $r(s_i, s_j) \in \{-1, 0, 1\}$ represents whether $s_i \prec s_j$, $s_i = s_j$ or $s_j \prec s_i$, respectively (Köppel et al. 2020b). While learning a pointwise ranking function is a regression problem, learning a pairwise ranking is a classification problem, i.e., given two states s_i and s_j, we need to assign it either -1, 0, or 1. As we show in the next section, this difference has a large impact on learning a ranking for planning in comparison to learning a heuristic.

3 Rank-based GBFS

In the context of GBFS, the heuristic function h is used solely for sorting states in the priority queue as shown in Line 4 of Algorithm 1. This means h is being used only as a total quasi-order over the states S. This fact has been exploited by others, for instance, by using functions other than the cost-to-goal to order states in GBFS (Ferber et al. 2022) and to analyse what makes a good heuristic for GBFS (Wilt and Ruml 2016). For this reason, we extend GBFS to use a ranking instead of a heuristic. Formally, given a total quasi-order $\preceq$, rank-based GBFS is the algorithm obtained by replacing Line 4 in Algorithm 1 with: choose s such that $s \preceq t$ for all $t \in \mathcal{O}$. We relate the two versions of GBFS with the lemma below.

Definition 2 ($\preceq_h$)**.** *Given a heuristic h, the total quasi-order $\preceq_h$ associated with it is so that $s \preceq_h s' \Leftrightarrow h(s) \leq h(s')$.*

Lemma 1 (Rank-based GBFS equivalence)**.** *Given a problem $\mathbb{P}$, a heuristic h, GBFS using h is equivalent to rank-based GBFS using $\preceq_h$ for the same tie-breaking method.*

It is easy to see that Lemma 1 holds based on the definition of $\preceq_h$. Lemma 1 also explains why any monotonic non-decreasing function f applied to the heuristic h, i.e., $h'(s) = f(h(s))$, does not change the solution found by GBFS: f preserves order, and therefore the total quasi-order $\preceq_h$ and $\preceq_{f(h)}$ are the same. We close this section by looking at what ranking function we should aim to learn.

An intuitive ranking to learn is the ranking induced by h^*, i.e., $\preceq_{h^*}$ since GBFS is guaranteed to find the optimal solution using h^* under unit-cost actions. However, for usage with GBFS, $\preceq_{h^*}$ orders more states than are necessary to find the optimal solution. To see this, consider the example in Figure 1. The total quasi-order $\preceq_{h^*}$ imposes an ordering between any pair of states in the example, including between siblings of states on the optimal path, i.e., $t_{i,j} \preceq_{h^*} t_{i',j'} \Leftrightarrow h^*(t_{i,j}) \leq h^*(t_{i',j'})$. However, the ordering among siblings is not needed, i.e., it is sufficient to know that $s_i \prec t_{j,m}$ for $j \in [\![i]\!]$ and $m \in [\![k_i]\!]$. We exploit this fact to define an optimal ranking as:

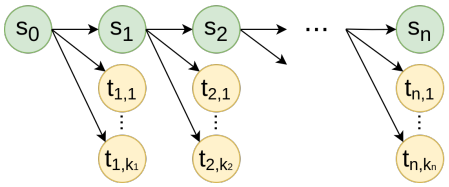

Figure 1: A example graph representation of part of the search tree. The optimal path is $[s_0, s_1, \cdots s_n]$. States $t_{i,j}$ are siblings of s_i and have higher ranks, i.e., $s_i \preceq t_{i,j}$.

Definition 3 (Optimal Ranking $\preceq^*$)**.** *Given a problem $\mathbb{P}$ and an optimal plan $\pi^* = [s_0, s_1, ..., s_n]$ for $\mathbb{P}$, the optimal ranking $\preceq^*$ is a total quasi-order such that $s_i \prec^* s_{i-1}$ for all $i \in [\![n]\!]$ and $s_i \prec^* t \; \forall t \in \mathsf{N}_{s_{i-1}}$ and $\forall i \in [\![n]\!]$.*

Notice that Definition 3 only relates a state s_i in the optimal plan with its parent s_{i-1} and its siblings t. Since it does not enforce an ordering between the siblings t of s_i, the optimal ranking is still a total quasi-order. Moreover, due to the transitivity property of total quasi-orders, we have that $s_i \prec^* t'$ for all $t' \in \cup_{j=0}^{i-1} \mathsf{N}_{s_j}$, i.e., s_i is ranked strictly better than any of its ancestors and their siblings. Next, we prove that GBFS finds the optimal solution when using an optimal ranking.

Theorem 1. *For a solvable problem $\mathbb{P}$ with unit costs, given an optimal ranking $\preceq^*$, the plan returned by the rank-based GBFS is optimal.*

Proof. Consider a problem $\mathbb{P}$ with unit costs, with an optimal solution $\pi^* = [s_0, s_1, ..., s_n]$ and optimal ranking $\preceq^*$. We prove by induction that s_{i-1} is expanded at the ith iteration of the algorithm and that after expansion, the open list is $\mathcal{O}_i = \cup_{j=0}^{i-1} \mathsf{N}_{s_j} \setminus \cup_{j=0}^{i-1} \{s_j\}$ for all $i \in [\![n]\!]$. For readability, given a state s and set of states X, we abbreviate $\forall t \in X, s \prec^* t$ with $s \prec^* X$. For the **base case:** initially, the open list of GBFS only contains s_0 (Line 1 of Algorithm 1), which is always expanded at the first iteration of the while loop in Line 3, leading to $\mathcal{O}_1 = \mathsf{N}_{s_0} \setminus \{s_0\}$. For the **induction step:** assume the property holds, we show that s_i is expanded next and $\mathcal{O}_{i+1} = \cup_{j=0}^{i} \mathsf{N}_{s_j} \setminus \cup_{j=0}^{i} s_j$. According to the definition of the optimal ranking we have: $s_i \prec^* \mathsf{N}_{s_{i-1}}$ (a) and $s_i \prec^* s_{i-1}$ (b). Because s_{i-1} was expanded in the last iteration, we have $s_{i-1} \prec^* \cup_{j=0}^{i-2} \mathsf{N}_{s_j} \setminus \cup_{j=0}^{i-2} \{s_j\}$ (c). Since $\prec^*$ is transitive, (b) and (c) lead to $s_i \prec^* \cup_{j=0}^{i-2} \mathsf{N}_{s_j} \setminus \cup_{j=0}^{i-2} \{s_j\}$ (d). Finally, the union of (a) and (d) gives $s_i \prec^* \cup_{j=0}^{i-1} \mathsf{N}_{s_j} \setminus \cup_{j=0}^{i-1} \{s_j\}$. In other words, s_i is ranked at the top of the open list O_i and will be expanded in iteration $i+1$. After expansion the queue will be $O_{i+1} = O_i \cup (N_{s_i} \setminus \cup_{j=0}^{i} \{s_j\}) = \cup_{j=0}^{i} N_{s_j} \setminus \cup_{j=0}^{i} \{s_j\}$. □

A key difference between optimal rankings and $\preceq_{h^*}$ is that there are several total quasi-orders that are optimal rankings while $\preceq_{h^*}$ is unique. This makes learning an optimal ranking potentially easier than learning $\preceq_{h^*}$ or even h^* itself, since h^* is also unique. We exploit this in the next section where we introduce our framework to learn an optimal ranking.

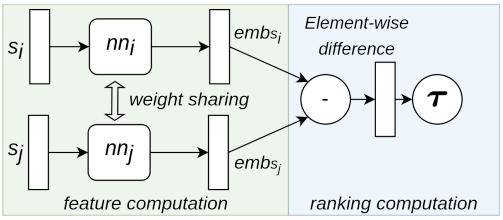

Figure 2: The structure of the DirectRanker (Köppel et al. 2020b).

4 Learning an Optimal Ranking

In this section, we introduce a new framework capable of leveraging any neural network regression model to learn an optimal ranking function. We also show how to efficiently integrate the ranking computations into GBFS and how to extract more data from the optimal plans for training while still efficiently handling this extra data through batching.

4.1 DirectRanker

Our approach for learning an optimal ranking through pairwise classification uses *DirectRanker* (Köppel et al. 2020b). DirectRanker is a general model for pairwise ranking that is designed to learn total quasi-orders. Its structure is shown in Figure 2. The model consists of two parts: feature computation and ranking computation. The first part receives two objects (in our case, states) s_i and s_j to be ranked and generates the embedding vectors emb_{s_i} and emb_{s_j}, both in $\mathbb{R}^n$, through the neural networks nn_i and nn_j. The structure of the neural networks nn_i and nn_j is not restricted; the only requirement is that both nn_i and nn_j have the same structure and share weights, i.e., they must be identical.

The second part of the model, ranking computation, computes the element-wise difference of the two embedding vectors and passes it through a single neuron τ with zero bias. Formally, this second part computes $p = \sigma(\mathbf{w}(emb_{s_i} - emb_{s_j}))$ where $\mathbf{w} \in \mathbb{R}^{1\times n}$ are the weights and σ is the activation function of τ. The activation function σ must be odd and sign-preserving, i.e., $\sigma(-x) = -\sigma(x)$ and $sign(\sigma(x)) = sign(x)$. The input states s_i and s_j are classified as $s_i \prec^* s_j$, $s_i = s_j$, or $s_j \prec^* s_i$ if p is less than, equal to, or greater than 0, respectively. Under these assumptions, DirectRanker is guaranteed to return a total quasi-order (Köppel et al. 2020b, Theorem 1). Besides the activation function σ, the loss function used by DirectRanker is also a parameter of the framework and we use the same parametrisation as in Köppel et al. (2020b) experiments: activation function $\sigma(x) = (1 + e^{-x})^{-1} - 0.5$ and mean squared error (MSE) as the loss function (Köppel et al. 2020a).

4.2 Efficient Integration with GBFS

Once we learn the shared weights for nn_1 and nn_2 as well as $\mathbf{w}$ for the ranking computation, we need to integrate it into GBFS. Let $D(s,t) \in \{-1, 0, 1\}$ represent the result of the DirectRanker for s and t. In order to insert a new state s into the priority queue $\mathcal{O}$, we need to compare s with states already in $\mathcal{O}$ since we have learned a pairwise ranking. However, the amortised computational complexity of insertion is $O(k \log |\mathcal{O}|)$, where k is the time needed to evaluate $D(s,t)$ for a single comparison.

Since the ranking computation part of DirectRanker is a single-layer, unbiased neural network, we can transform our learned pairwise ranking into a point-wise representation to improve the insertion efficiency in the priority queue. This transformation is formalised in Lemma 2 and lets us represent the learned $D(s,t)$ as the point-wise ranking $r(s) \in \mathbb{R}$. As a result, the priority queue in GBFS can now be sorted by $r(s)$ resulting in an amortised insertion time of $O(k' + \log |\mathcal{O}|)$ where k' is the time required to evaluate $r(s)$. With this transformation, only a single neural network evaluation is needed instead of $\log |\mathcal{O}|$ when using the pairwise ranking function $D(s,t)$. Although $r(s)$ is being used as a heuristic in GBFS, it is important to note that it is not a heuristic in the traditional sense, for instance, $r(s)$ can be negative. Moreover, since we are learning an optimal ranking, it is possible that $r(t) \leq r(t')$ even though $h^*(t) > h^*(t')$ for states t and t' outside the optimal plan.

Lemma 2. *Given a DirectRanker model with weights* $\mathbf{w}$ *representing the total quasi-order* $\preceq$*, we can extract a point-wise ranking function* $r : \mathsf{S} \to \mathbb{R}$ *representing* $\preceq$*.*

Proof. Given two states $s_i, s_j \in \mathsf{S}$, without loss of generality, we assume $s_i \preceq s_j$. Then from the definition of the DirectRanker, we have $p = \sigma(\mathbf{w} \cdot (nn_1(s_i) - nn_2(s_j))) \leq 0$. Since nn_1 and nn_2 are identical and σ is sign-preserving, we have that $p \leq 0 \Leftrightarrow \mathbf{w} \cdot (nn(s_i) - nn(s_j)) \leq 0$. Because the right-hand side is a linear transformation, it is equivalent to $\mathbf{w} \cdot nn(s_i) \leq \mathbf{w} \cdot nn(s_j)$. Defining $r(s)$ as $\mathbf{w} \cdot nn(s)$, we have that $p \leq 0 \Leftrightarrow r(s_i) \leq r(s_j)$ and therefore $s_i \preceq s_j \Leftrightarrow r(s_i) \leq r(s_j)$. □

4.3 Efficient Training

Another key difference of our framework lies in the training methodology. Unlike when learning a heuristic, where a specific value is required for each state, we train our DirectRanker $D(s_i, s_j)$ to learn the optimal ranking through classification and then extract the point-wise ranking function $r(s)$. Thus, given a pair of states s_i and s_j, it is sufficient to determine their relative optimal ranking, i.e., whether $s_i \preceq^* s_j$ or $s_j \preceq^* s_i$, without assigning specific values to each state. This allows us to use siblings of states in optimal plans in the training process without the need to compute any information about them. Thus, for a problem with a constant branching factor b, an optimal plan π^* of size n encodes $O(bn)$ optimal ranking pairs for training. This contrasts with methods that directly learn a heuristic function (e.g., STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020) and GOOSE (Chen, Thiébaux, and Trevizan 2024)), which can only extract n pairs in the form $(s, h^*(s))$ from π^*. Note that it is computationally expensive to compute $h^*(s')$ for any $s' \notin \pi^*$ since it requires optimally solving the problem from s'.

Another advantage of our framework is that we can efficiently handle this extra training data by batching the pairs $s_i \prec t$ for all siblings t of s_i and also $t = s_{i-1}$ for each $s_i \in$

domain	Training		Testing	
blocksworld	$n \in [2, 21]$	56	$n \in [5, 488]$	90
childsnack	$c \in [1, 5]$	37	$c \in [4, 292]$	90
ferry	$c \in [1, 13]$	66	$c \in [2, 974]$	90
floortile	$t \in [2, 30]$	64	$t \in [12, 980]$	90
miconic	$p \in [1, 10]$	99	$p \in [1, 485]$	90
rovers	$r \in [1, 4]$	67	$r \in [1, 30]$	90
satellite	$s \in [1, 10]$	90	$n \in [3, 99]$	90
sokoban	$n \in [7, 13]$	99	$b \in [8, 99]$	90
spanner	$s \in [1, 10]$	89	$n \in [1, 487]$	90
transport	$v \in [1, 5]$	47	$n \in [3, 50]$	90

Table 1: Number of problems per domain and problem size in terms of the most significant object type: n blocks in blocksworld, c children in childsnack, c cars in ferry, etc. The full set of parameters governing problem size can be found in the (Segovia and Seipp 2023).

π^* (Damke and Hüllermeier 2021). The performance gains come from the fact that the embedding emb_{s_i} is computed only once as opposed to $|\mathsf{N}_{s_{i-1}}|$ times, i.e., once for each pair involving s_i. Formally, let $B_{s_i} = \{s_{i-1}\} \cup (\mathsf{N}_{s_{i-1}} \setminus \{s_i\})$, i.e., the set consisting of the siblings of s_i and its parent. We compute the embeddings for all states in B_{s_i} resulting in the matrix $\mathbf{E}_r \in \mathbb{R}^{m \times n}$ where $m = |B_{s_i}|$ and the j-th row of $\mathbf{E}_r$ is the vector embedding of the j-th element in B_{s_i}. Next, we compute the embedding emb_{s_i} and stack m copies of it to create the matrix $\mathbf{E}_l \in \mathbb{R}^{m \times n}$. Then we compute $\mathbf{p} = \sigma((\mathbf{E}_l - \mathbf{E}_r)\mathbf{w}) \in \mathbb{R}^m$ where the j-th row of $\mathbf{p}$ is the output of τ comparing s_i and the j-th element in B_{s_i}. Thus, we compute $\mathbf{p}$ by generating only $|B_{s_i}| + 1$ embeddings as opposed to $2|B_{s_i}|$ embeddings. This difference is important since computing each embedding in our experiments requires evaluating an expensive neural network architecture.

5 Related Work

The closest work to ours is (Chrestien et al. 2023) which learns a heuristic function using a pointwise ranking as the target, as opposed to h^*. As with other approaches that learn a heuristic function, e.g., STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020) and GOOSE (Chen, Thiébaux, and Trevizan 2024), their learned heuristics can be used with A*. However, all these approaches cannot guarantee that the solution found is optimal since the learned heuristic may not be admissible. For this reason, our work focuses on GBFS which allows us to remove the restriction that the learned ranking must also be a heuristic.

What makes the approach by Chrestien et al. (2023) different from all previous works on learning heuristics is that their target is a *perfect ranking*, a concept they introduced. A perfect ranking orders a pair of states s and t as $s \prec t$ for all s on the optimal path and $t \in \mathcal{O}$ when s is popped from the GBFS queue. It is equivalent to an optimal ranking without transitivity nor the requirement that $s_i \prec s_{i-1}$ for two consecutive states s_{i-1} and s_i in π^*. To see the benefits of using an optimal ranking as the target instead of a perfect ranking, consider a problem with a constant branching factor b and an optimal plan $\pi^* = [s_0, s_1, ..., s_n]$. The number of ordered pairs $s \prec t$ encoded in the optimal plan is $\sum_{i=1}^{n} ib = b(n^2 + n)/2$. To learn a perfect ranking, Chrestien et al. (2023) use all pairs encoded in the optimal plan with the exception of the pairs $s_i \prec s_j$ for $s_i, s_j \in \pi^*$ and $i < j$, resulting in $(b-1)(n^2+n)/2$ total pairs. Alternatively, due to transitivity and the requirement that $s_i \prec s_{i-1}$ for $s_i \in \pi^*$ imposed by our optimal ranking, we can encode all the $b(n^2 + n)/2$ pairs using only bn pairs. Therefore, our training data scales linearly instead of quadratically with $|\pi|$ while representing all ordered pairs considered by (Chrestien et al. 2023). As our experiments will show, this has a large empirical effect on training efficiency and generalisation.

Our approach also differs in how the target ranking is learned. We learn a pairwise ranking function using classification through the DirectRanker architecture that is later converted to a pointwise ranking to be efficiently used with GBFS. The heuristic learned by (Chrestien et al. 2023) follows an approach in which a pointwise ranking function is learned using the 0-1 loss for an implicit pairwise ranking function. Their loss function is later relaxed to a logistic loss in order to use gradient optimisation.

Another work based on ranking is (Garrett, Kaelbling, and Lozano-Pérez 2016). They learn a pairwise ranking by training a Rank Support Vector Machine model (Joachims 2002) using a convex relaxation of the *Kendall rank correlation coefficient*. This coefficient measures the degree of similarity between two rankings by computing the difference between the number of concordant and discordant pairs as a proportion of the total number of possible pairs. One main difference between our approaches is that we allow the use of any neural network for computing features while they use handcrafted features extracted from the graph implicitly built by the FF heuristic (Hoffmann and Nebel 2001). Another key difference is that they use as data only the pairs within a suboptimal plan π, that is, they do not use the states outside of π.

6 Experiments

In this section, we empirically compare our ranking framework to Chrestien et al. 2023's framework using GBFS. For both frameworks, we consider two state-of-the-art neural network regression models, namely STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020) and GOOSE (Chen, Thiébaux, and Trevizan 2024), for internal feature computation. For the latter one, we experiment with two different types of graph, i.e. the ILG (Chen, Trevizan, and Thiébaux 2024) and LLG (Chen, Thiébaux, and Trevizan 2024). To verify that learning a ranking is beneficial, we consider STRIPS-HGN and GOOSE by themselves, i.e., trained using h^* as the learning target. As a baseline, we also consider the h^{FF} heuristic (Hoffmann and Nebel 2001).

6.1 Dataset

We use the 10 domains from the 2023 International Planning Competition Learning Track (IPC23-LT) (Segovia and Seipp 2023). Each domain provides 100 training problems

and 90 testing problems. We first solve the training problems using the Scorpion planner (Seipp, Keller, and Helmert 2020) with a 30-minute time limit. Only problems solved optimally by Scorpion are used for training, but this still results in a dataset of sufficient size for our experiments. The training problems are randomly split between the training set (90%) and the validation set (10%). The problems in the validation set are used only to monitor convergence and control learning rates.

The testing set is divided into "easy", "medium" and "hard" subsets per domain, based on problem size, with 30 problems per subset. In contrast the training set only contains "easy" problems. Details of the dataset are shown in Table 1.

6.2 Training Settings

The three graph neural network regression models we use are GOOSE with the Instance Learning Graph (ILG) (Chen, Trevizan, and Thiébaux 2024), GOOSE with the Lifted Learning Graph (LLG) (Chen, Thiébaux, and Trevizan 2024) and STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020). The LLG encodes both the planning problem and domain in a lifted form, whereas the ILG only encodes the former. STRIPS-HGN encodes the problem delete relaxation in a grounded form. All models can learn domain-independent heuristics and our framework inherits this property. However, our experiments are restricted to domain-specific rankings.

For all regression models, we used the code provided by their authors with all ReLU activations replaced by LeakyReLUs, except in their last layer. The ReLU function is suitable for learning heuristics because its values are non-negative; however, this restriction is not needed for learning a ranking and we observed that it results in slower convergence. For consistency, the same model parameters are used across our experiments: 4 message-passing layers and hidden dimension $m = 64$. For our framework, we use the output from the second-last hidden layer of the underlying regression model as the embedding vector, thus $emb_s \in \mathbb{R}^{64}$.

For each domain, we compare GBFS using the following configurations for the heuristic/ranking:

ILG, LLG and HGN. The original GOOSE-ILG, GOOSE-LLG and STRIPS-HGN regression models trained using h^* as target and their original loss function. The training dataset contains all state-value pairs $(s, h^*(s))$ for s in the optimal plan.

OptRank(ILG), OptRank(LLG) and OptRank(HGN). Our optimal ranking framework combined with the respective regression models. The training dataset is the set of all ordered pairs $s \prec t$ for s in the optimal plan and $t \in \mathsf{N}_{s'} \cup \{s'\}$ where s' precedes s in the optimal plan.

PerfRank(ILG), PerfRank(LLG) and PerfRank(HGN). Chrestien et al. (2023)'s framework to learn a perfect ranking heuristic combined with each model. The loss function associated with the perfect ranking definition is also used. The training dataset consists of all ordered pairs $s \prec t$ for s in the optimal plan and $t \in \mathcal{O}$ when s is popped from the GBFS queue.

We use the Adam optimiser and an initial learning rate of 10^{-3}. The learning rate is reduced by a factor of 10 if the accuracy on the validation set does not improve for 10 consecutive epochs. Training is stopped when the learning rate reaches 10^{-6} or after 500 epochs.

For each configuration, we solve the problems in the testing set with Fast-Downward using eager-GBFS (Helmert 2006), a time limit of 30 min per problem, and 8 GB of memory. All experiments are run on an Intel Xeon 2.1GHz CPU and NVIDIA A6000 GPU with 64GB of memory. The experiments were repeated three times and the average results are reported. The source code can be found at (Hao, Thiébaux, and Trevizan 2024) and an extended set of results in (Hao et al. 2024).

6.3 Results

Total Coverage. Table 2 shows the coverage of each configuration on each domain, i.e., the number of problems it solves for this domain. Overall, using optimal rankings improved the coverage of the underlying regression model: 15%, 22% and 32% increase for GOOSE-ILG, GOOSE-LLG and STRIPS-HGN, respectively. Our optimal ranking approach also obtained higher total coverage than the perfect ranking one: 7%, 8% and 176% increase for GOOSE-ILG, GOOSE-LLG and STRIPS-HGN, respectively. The large increase in coverage obtained by OptRank(HGN) w.r.t PerfRank(HGN) can be partially attributed to: (i) STRIPS-HGN being a slower model to train; and (ii) PerfRank(HGN) using quadratically more data than OptRank(HGN). The convergence time for all learned configurations is shown in Figure 3 and we observe that, for any framework, using STRIPS-HGN to generate features is slower than using GOOSE.

The impact of using STRIPS-HGN is less pronounced on OptRank(HGN) due to its compact dataset that exploits transitivity and is nearly equivalent to that of PerfRank(HGN). Moreover, despite processing $O(b)$ more data than LLG where b is the average branching factor of a domain, OptRank(LLG) converges in a similar amount of time. However, OptRank(ILG) converges slower than ILG, but the marginal increase in time to convergence is much smaller than that of PerfRank(ILG). Hence, it processes training data much faster.

Domain-specific coverage. When considering individual domains, OptRank(LLG) obtains the highest coverage in two domains (Childsnack and Transport) and OptRank(ILG) in one domain (Blocksworld), with 15% (4 problems), 12% (5 problems) and 35% (18 problems) more coverage than the second-best planner, respectively. PerfRank(LLG) has the highest coverage only on Spanner, where it solves one more problem (2%) than OptRank(ILG), PerfRank(ILG) and OptRank(LLG). In Floortile, all configurations perform very poorly. This domain is known to be hard because it requires a large receptive field and it has many state space symmetries.

Model	Domain	blocks-world	child-snack	ferry	floortile	miconic	rovers	satellite	sokoban	spanner	transport	sum
ILG	easy	30	18	30	0	30	25	26	27	30	30	246
	med.	20	0	30	0	30	1	1	1	6	9	98
	hard	0	0	1	0	16	0	0	0	0	0	17
	sum	50	18	61	0	76	26	27	28	36	39	361
PerfRank (ILG)	easy	30	16	30	2	30	28	28	20	30	28	242
	med.	27	3	30	0	30	0	0	1	30	2	123
	hard	5	0	3	0	16	0	0	0	1	0	25
	sum	62	19	63	2	76	28	28	21	61	30	390
OptRank (ILG)	easy	30	21	30	1	30	28	30	30	30	30	260
	med.	30	1	30	0	30	0	5	2	30	1	129
	hard	9	0	3	0	16	0	0	0	1	0	29
	sum	**69**	22	63	1	76	28	35	32	61	31	418
LLG	easy	30	19	30	1	30	29	27	26	30	30	252
	med.	0	0	30	0	30	2	2	1	4	8	77
	hard	0	0	1	0	15	0	0	0	0	0	16
	sum	30	19	61	1	75	31	29	27	34	38	345
PerfRank (LLG)	easy	30	21	30	2	30	28	27	30	30	30	258
	med.	5	4	30	0	30	1	0	2	30	9	111
	hard	0	0	2	0	16	0	0	0	2	0	20
	sum	35	25	62	2	76	29	27	32	**62**	39	389
OptRank (LLG)	easy	30	26	30	1	30	30	29	30	30	30	266
	med.	20	4	30	0	30	1	4	2	30	16	137
	hard	1	0	1	0	15	0	0	0	1	0	18
	sum	51	**30**	61	1	75	31	33	32	61	**46**	421
HGN	easy	22	6	26	0	30	25	13	27	30	22	201
	med.	0	0	2	0	30	0	0	0	0	0	32
	hard	0	0	0	0	0	0	0	0	0	0	0
	sum	22	6	28	0	60	25	13	27	30	22	233
PerfRank (HGN)	easy	0	13	0	1	24	10	6	17	30	6	107
	med.	0	0	0	0	0	0	0	0	4	0	4
	hard	0	0	0	0	0	0	0	0	0	0	0
	sum	0	13	0	1	24	10	6	17	34	6	111
OptRank (HGN)	easy	23	25	30	1	30	28	29	30	30	28	254
	med.	0	0	8	0	27	0	2	1	14	0	52
	hard	0	0	0	0	1	0	0	0	0	0	1
	sum	23	25	38	1	58	28	31	31	44	28	307
hFF	sum	28	26	**68**	**12**	**90**	**34**	**65**	**36**	30	41	**430**

Table 2: Rounded average coverage per domain and problem difficulty for the different configurations. Best coverage per domain highlighted in bold. The best coverage for each regression model is underlined.

GOOSE-LLG configurations. OptRank(LLG) improves LLG's coverage in 6 domains and obtains the same coverage in the other 4 domains. Therefore, using optimal rankings is highly beneficial for this regression model. E.g, in Blocksworld and Spanner, LLG solves none or hardly any of the medium-difficulty problems, whereas OptRank(LLG) solves a large fraction or all of them, demonstrating a greater ability to generalise to larger problem sizes. PerfRank(LLG) also improves the coverage of LLG in almost all domains. The coverage is marginally smaller (2 problems, representing about 9%) only in Rovers and Satellite. OptRank(LLG) has better coverage than PerfRank(LLG) in 5 domains, namely Blocksworld, Childsnack, Rovers, Satellite and Transport, with 46% (16 problems), 20% (5 problems), 7% (2 problems), 22% (6 problems) and 18% (5 problems) more coverage, respectively. Their coverage is the same for Sokoban and PerfRank(LLG) solves one more problem than OptRank(LLG) for Ferry, Floortile, Miconic and Spanner.

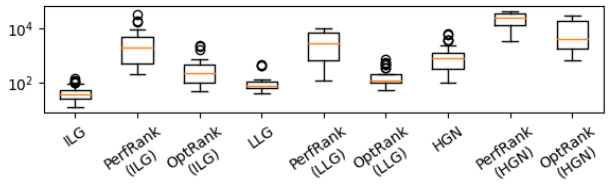

Figure 3: Convergence times (sec) grouped by configuration. Each box shows the training time of all 10 domains and 3 learned models for the configuration. PerfRank(HGN) Models whose training failed to converge are excluded.

GOOSE-ILG configurations. OptRank(ILG) improves ILG's coverage in 8 domains and has the same coverage on

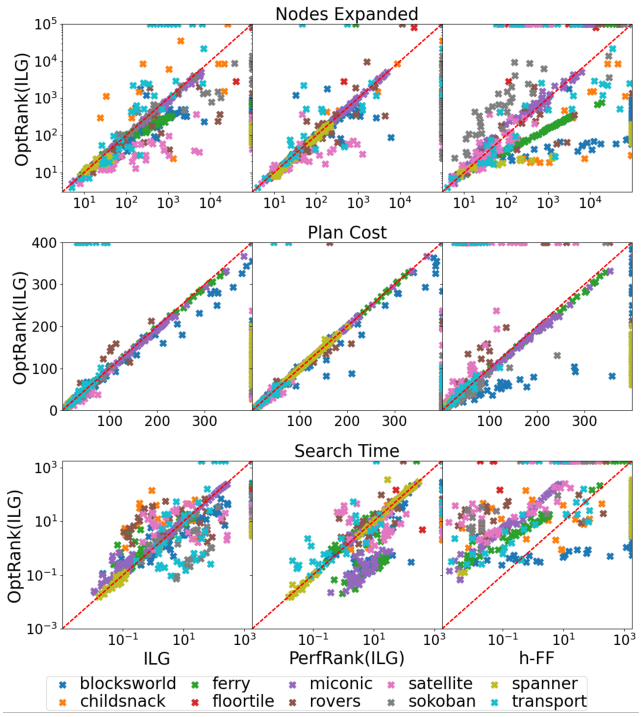

Figure 4: Comparison between OptRank(ILG) and ILG (first column), PerfRank(ILG) (second column), and h-FF (third column) w.r.t. the number of nodes expanded (first row), plan cost (second row) and search time (third column). Problems only solved by one of the planners are mapped to the plot's boundaries. Points in the lower-triangle parts favor OptRank(ILG).

Miconic. Although OptRank(ILG) shows 20% (8 problems) worse coverage on Transport, the marginal improvements on other domains are more significant, e.g. Blocksowrd, Satellite, Spanner with 38% (19 problems), 30% (8 problems) and 69% (25 problems) more coverage, respectively. Therefore, using optimal rankings is beneficial overall. OptRank(ILG) has better coverage than PerfRank(ILG) in 5 domains, namely Blocksworld, Childsnack, Satellite Sokoban and Transport, with 11% (7 problems), 10% (2 problems), 25% (7 problems), 50% (11 problems) and 3% (1 problem) more coverage, respectively. Their coverage is the same for Ferry, Miconic, Rovers and Spanner. PerfRank(ILG) solves one more problem than OptRank(ILG) for Floortile.

To better understand the differences between OptRank(ILG) and both ILG and PerfRank(ILG), Figure 4 compares them regarding the number of nodes expanded, the cost of the returned plan and the search time to find a plan. We observe that OptRank(ILG) generally expands fewer nodes than ILG while achieving similar plan quality. This means OptRank(ILG) spends less time exploring unpromising regions of the search space and shows that our learned ranking generalises better than the learned ILG heuristic to unseen problems. Although our framework guarantees to learn a total quasi-order, there is no guarantee that the learned quasi-order is an *optimal* ranking. This can be observed in the plan quality plots where other approaches find plans cheaper than OptRank(ILG) for some problems. We also observe that PerfRank(ILG) generally expands more nodes than OptRank(ILG) while providing plans of the same or slightly lesser quality.

STRIPS-HGN configurations. Considering only configurations based on STRIPS-HGN, OptRank(HGN) has the highest coverage in all domains except in Miconic where it solves two fewer problems than HGN. As with GOOSE, using optimal rankings is also highly beneficial for this regression model. Surprisingly, PerfRank(HGN)'s coverage is lower than HGN's except on Childsnack, Floortile and Spanner. Moreover, PerfRank(HGN) did not converge for Blocksworld and Ferry. Note that our experiments use the original version of STRIPS-HGN (Shen, Trevizan, and Thiébaux 2020) whereas Chrestien et al. (2023) use an improved version of STRIPS-HGN. This and the different benchmark problems for each domain could explain part of the discrepancy.

Comparison with h^{FF}. Finally, none of the learned models were able to surpass the coverage of h^{FF}. However, OptRank(LLG) is not far off with just 9 fewer problems solved. This is an improvement on the competition results where the deep learning entries attempting to directly learn a heuristic or a policy did not exceed the score of HGN in our experiments. Moreover, OptRank(ILG) generally expands fewer nodes and finds plans of better quality than h^{FF}. Hence it is the inference time of GNN models remains an impediment to their competitiveness with special-purpose planning heuristics.

7 Conclusion

Heuristic search algorithms conventionally assume heuristic functions mapping states to goal-distance estimates. Yet, as was previously observed, for the popular greedy best-first search algorithm, the heuristic function merely serves to rank states, which is an overkill. To find a plan without search, as we show here, we only require an "optimal" ranking relating the states on an optimal plan to its neighbours. While this observation has yet to be used to design model-based heuristics, as we show here it is highly relevant for learning, and enables us to obtain much better performance than heuristic functions learned using the same neural network representations – even compared to a closely related ranking-learning framework.

It remains of course a bit disappointing that, at least in the benchmarks used here, the classic h^{FF} heuristic still performs best overall. This may change though with further research into representations and training algorithms for learning rankings. Apart from this, our results raise interesting questions regarding learning search guidance information for other settings and algorithms, for instance: Can our approach be extended to non-unit-cost problems? Are there other algorithms (e.g., hill-climbing or beam search) in which the heuristic function can be replaced with a simpler function that can be efficiently learned? What about variants of greedy best-first search for other forms of planning, such as top-K?

Acknowledgements

We thank the reviewers for their comments which helped improving the paper. This work was supported by Australian Research Council grant DP220103815, by the Artificial and Natural Intelligence Toulouse Institute (ANITI) under the grant agreement ANR-19-PI3A-0004, and by the European Union's Horizon Europe Research and Innovation program under the grant agreement TUPLES No. 101070149.

References

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *Artificial Intelligence*, 175(16-17): 2075–2098.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A Robust and Fast Action Selection Mechanism for Planning. In *Proc. National Conference on Artificial Intelligence (AAAI)*, 714–719.

Burges, C.; Shaked, T.; Renshaw, E.; Lazier, A.; Deeds, M.; Hamilton, N.; and Hullender, G. 2005. Learning to Rank Using Gradient Descent. In *Proc. International Conference on Machine learning (ICML)*, 89–96.

Burges, C. J. 2010. From RankNet to LambdaRank to LambdaMART: An Overview. Technical Report MSR-TR-2010-82, Microsoft Research.

Chen, D. Z.; Thiébaux, S.; and Trevizan, F. 2024. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, 38(18): 20078–20086.

Chen, D. Z.; Trevizan, F.; and Thiébaux, S. 2024. Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. In *34th International Conference on Automated Planning and Scheduling (ICAPS)*.

Chrestien, L.; Edelkamp, S.; Komenda, A.; and Pevny, T. 2023. Optimize Planning Heuristics to Rank, Not to Estimate Cost-to-Goal. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 25508–25527.

Cohen, E.; and Beck, J. C. 2018. Local Minima, Heavy Tails, and Search Effort for GBFS. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 4708–4714.

Damke, C.; and Hüllermeier, E. 2021. Ranking Structured Objects with Graph Neural Networks. In *Proc. International Conference on Discovery Science (DS)*, 166–180.

Ferber, P.; Geiber, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2022. Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods. *Proc. International Conference on Automated Planning and Scheduling (ICAPS)*, 32(1): 583–587.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *Proc. European Conference on Artificial Intelligence (ECAI)*, 2346–2353.

Freund, Y.; Iyer, R.; Schapire, R. E.; and Singer, Y. 2003. An Efficient Boosting Algorithm for Combining Preferences. *Journal of Machine Learning Research*, 4: 933–969.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 3089–3095.

Hao, M.; Thiébaux, S.; and Trevizan, F. 2024. Source Code for "Guiding GBFS Through Learned Pairwise Rankings". https://doi.org/10.5281/zenodo.11107790.

Hao, M.; Trevizan, F.; Thiébaux, S.; Ferber, P.; and Hoffmann, J. 2024. Learned Pairwise Rankings for Greedy Best-First Search. In *Proc. ICAPS Workshop on Reliable Data-Driven Planning and Scheduling*.

Heller, D.; Ferber, P.; Bitterwolf, J.; Hein, M.; and Hoffmann, J. 2022. Neural Network Heuristic Functions: Taking Confidence Into Account. In *Proc. International Symposium on Combinatorial Search (SOCS)*, 223–228.

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

Herbrich, R.; Graepel, T.; and Obermayer, K. 1999. Large Margin Rank Boundaries for Ordinal Regression. In *Advances in Large Margin Classifiers*. The MIT Press.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Joachims, T. 2002. Optimizing Search Engines Using Clickthrough Data. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 133–142.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, 8064–8073.

Köppel, M.; Segner, A.; Wagener, M.; Pensel, L.; Karwath, A.; and Kramer, S. 2020a. Code and Supplementary Material to the Paper: Pairwise Learning to Rank by Neural Networks Revisited: Reconstruction, Theoretical Analysis and Practical Performance. https://doi.org/10.5281/zenodo.3700819.

Köppel, M.; Segner, A.; Wagener, M.; Pensel, L.; Karwath, A.; and Kramer, S. 2020b. Pairwise Learning to Rank by Neural Networks Revisited: Reconstruction, Theoretical Analysis and Practical Performance. In *Proc. European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 237–252.

Li, H. 2022. *Learning to Rank for Information Retrieval and Natural Language Processing*. Springer.

Liu, T.-Y.; et al. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends® in Information Retrieval*, 3(3): 225–331.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Röger, G.; and Helmert, M. 2010. The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning. In *Proc. International Conference on Automated Planning and Scheduling (ICAPS)*, 246–249.

Segovia, J.; and Seipp, J. 2023. Benchmarking Repository of IPC 2023 - Learning Track. https://github.com/ipc2023-learning/benchmarks.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research*, 67: 129–167.

Shashua, A.; and Levin, A. 2002. Ranking with Large Margin Principle: Two Approaches. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 937–944.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. International Conference on Automated Planning and Scheduling (ICAPS)*, volume 30, 574–584.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proc. International Conference on Automated Planning and Scheduling (ICAPS)*, volume 32, 629–637.

Wilt, C. M.; and Ruml, W. 2016. Effective Heuristics for Suboptimal Best-First Search. *Journal of Artificial Intelligence Research*, 57: 273–306.

Wu, Q.; Burges, C. J.; Svore, K. M.; and Gao, J. 2010. Adapting Boosting for Information Retrieval Measures. *Information Retrieval*, 13: 254–270.