Inductive logic programming for heuristic search

Rojina Panta[1][0009−0006−9298−4431], Vedant Khandelwal[1][0000−0002−7291−3066], Celeste Veronese[2][0009−0007−7461−4039], Amit Sheth[1][0000−0002−0021−5293], Daniele Meli[2][0000−0002−3162−388X], and Forest Agostinelli[1][0000−0003−1392−3667]

[1] Department of Computer Science and Engineering, University of South Carolina, Columbia SC 29201, USA
{rpanta,vedant}@email.sc.edu, foresta@cse.sc.edu, amit@sc.edu
[2] Department of Computer Science, University of Verona, Italy
{daniele.meli, celeste.veronese}@univr.it

Abstract. Pathfinding problems are found through computing, chemistry, mathematics, and robotics. Solving pathfinding problems is typically achieved through heuristic search, which is guided by a heuristic function that can be learned using deep neural networks. However, since deep neural networks are typically not explainable, the extraction of new knowledge from these learned heuristic functions is cumbersome. On the other hand, to the best of our knowledge, it has yet to be shown how heuristic functions represented as logic programs, which have been shown to be explainable, can be learned. In this work, we present an algorithm to learn heuristic functions represented as logic programs using dynamic programming and inductive logic programming. Furthermore, we build on dynamic programming concepts to improve the learned logic programs by reusing predicates learned for solving simpler pathfinding problem instances to solve more complex instances. We use the 8-puzzle to demonstrate the effectiveness of our algorithm.

Keywords: Heuristic functions · Pathfinding · Reinforcement learning · Inductive logic programming.

1 Introduction

Pathfinding problems are a class of sequential decision making problems where the objective is to find a sequence of actions (i.e., a path) that transforms a start state into a goal state. Heuristic search is one of the most prominent methods for solving pathfinding problems such as puzzle solving [13, 14], chemical synthesis [6], and quantum circuit synthesis [25]. A core component of heuristic search is the heuristic function, which maps states to the estimated cost to reach the closest goal state from the given state via a shortest path, also known as the "cost-to-go". While constructing an informative heuristic function may require significant domain-specific knowledge, research has shown that informative domain-specific heuristic functions can be automatically constructed using deep reinforcement learning (RL) [1]. However, since deep RL uses RL [28] to train a heuristic function represented as a deep neural network (DNN) [26], the

learned heuristic function is not explainable. In this paper, we seek to show that inductive logic programming (ILP) [18, 7] can be combined with heuristic search to train a heuristic function represented as a logic program.

The ability to automatically construct an informative domain-specific heuristic function from just a description of a pathfinding problem allows one to solve pathfinding problems without having to manually acquire the necessary domain-specific knowledge. While this can lead to rapid advancements in fields with pathfinding problems, this domain-specific knowledge could be relevant to other open problems in related fields. For example, heuristic information for the cheapest way to synthesize certain molecules could be relevant to the importance of molecular functional groups for treating other diseases. While extracting such knowledge from DNNs has proven difficult, it has been shown that humans can learn from logic programs [20, 3, 29]. Therefore, representing a heuristic function as a logic program opens up new possibilities for learning new information about pathfinding problems, and ILP allows us to learn these domain-specific heuristic functions.

2 Background

2.1 Pathfinding

The objective of pathfinding is to find a sequence of actions (e.g., a path) that transforms a start state into a goal state. Formally, a pathfinding problem is defined as a weighted directed graph [24], where nodes represent states, edges represent actions that transition between states, weights on the edges represent transition costs, a given node represents the start state, and a given set of nodes represents the goal states. T represents the transition function where $s' = T(s, a)$ for some action, a, corresponds to an edge that connects s and s' in the weighted directed graph. c represents the transition cost function where $c(s, a)$, the cost of taking action a in state s, corresponds to the weight on the edge from s for action a. The cost of a path is the sum of transition costs and solutions with a lower path cost are preferred over those with a higher path cost.

2.2 Heuristic Search

Heuristic search is a widely used approach for solving pathfinding problems. Heuristic search uses a heuristic function, which maps states to their estimated cost-to-go, to guide a search from a given start state to a given goal. One of the most notable heuristic search algorithms, A* search [10], maintains a search tree, where nodes represent states and edges represent actions. The path cost of a node is the sum of transition costs from the start node to that node and its heuristic value is the output of heuristic function when applied to the state associated with that node. A node is expanded by applying every possible action to the state associated with that node and creating a child node from the resulting states. A* search maintains a priority queue of leaf nodes in the search tree,

where nodes with lower cost get higher priority. The cost of a node is the sum of its path cost and its heuristic value. A* search also maintains a dictionary, called "CLOSED", that maps states that have been seen during the search to the shortest path found to that state. A* search iteratively selects the node with the lowest cost from the priority queue, expands the node, and evaluates child nodes with the heuristic function. A* search only puts child nodes in the priority queue if their associated state is not in CLOSED or has been reached via a shorter path. A* search terminates when a node associated with a goal state is selected for expansion and returns the path to that node.

2.3 Learning Heuristic Functions with Deep Reinforcement Learning

On a high level, the algorithm to learn heuristic functions with deep reinforcement learning and using our algorithm for logic programming follows Algorithm 1. In domains with reversible actions, such as the 8-puzzle, states are generated by taking actions in reverse from the goal. The main differences come in the step to compute updated heuristic values and the step to update the heuristic functions based on states and their updated heuristic values. For deep reinforcement learning, computing updated heuristic values is done using value iteration and updating the heuristic function is done using gradient descent.

Approximate value iteration (AVI) [5], a dynamic programming [4] and core reinforcement learning algorithm, can be used to learn a heuristic function for heuristic search. The value iteration update, in the context of pathfinding, is shown in Equation 1, where $\mathcal{A}$ is the set of all possible actions. For large state spaces, this value iteration update can be approximated by a function with parameters, θ. When the function is a DNN, Equation 2 can be used as the loss function to train the parameters using gradient descent. In this equation, θ^- represents the parameters of a target network [16] whose parameters are periodically updated to θ to stabilize training in the presence of a non-stationary learning target. The DeepCubeA algorithm combined AVI with heuristic search to solve puzzles such as the Rubik's cube and Sokoban [1]. This approach has since been extended to pathfinding problems such as quantum algorithm compilation [25], cryptography [12], parking lot optimization [27], reaction mechanisms [23], and generalizing over goals [2].

$$h'(s) = \min_{a \in \mathcal{A}} c(s,a) + h(T(s,a)) \tag{1}$$

$$L(\theta) = \frac{1}{N} \sum_{i}^{N} \min_{a \in \mathcal{A}} \left(c(s_i,a) + h_{\theta^-}(T(s_i,a)) - h_\theta(T(s_i,a)) \right)^2 \tag{2}$$

2.4 Inductive Logic Programming

Inductive Logic Programming (ILP) is a symbolic machine learning framework that induces logical rules from structured data using a background theory and

a formal hypothesis space. A generic ILP task under a logical formalism F is defined as the tuple: $\mathcal{T} = \langle B, S_M, E \rangle$, where B is the *background knowledge*, a set of known logical statements (e.g., type constraints or static facts), S_M is the *search space*, i.e., the set of all axioms expressible in F that conform to a mode declaration M [19], and E is the set of *examples* to be covered by a learned hypothesis $H \subseteq S_M$. The objective is to find a hypothesis H such that $B \cup H \models E$, where $\models$ denotes logical entailment [17].

In this work, we perform ILP with Popper [8], a state-of-the-art ILP tool. Popper synthesizes logic programs with a learning-from-failure strategy, by progressively increasing the size of H when a conflict with one example arises (either $e \in E^+ \ : \ H \cup B \not\models e$ or $e \in E^- \ : \ H \cup B \models e$). The *optimal solution* found by Popper is then defined as the smallest hypothesis that, together with the background knowledge, entails all positive examples and does not entail any negative examples.

3 Methods

3.1 Heuristic Function Representation

The logic program heuristic function, h, is represented as a dictionary that maps cost-to-go values to logic programs. Therefore $h[c]$, for some cost-to-go, c, represents a logic program. States are represented as predicates in first-order logic, as they would be for examples in an inductive logic programming task. The heuristic value for a given state, s, is computed by returning the largest cost-to-go whose program entails that state. That is, the largest c such that $h[c] \cup B \models s$. If no c is found then a value of 0 is returned. We find this c using binary search (see Appendix). When learning, the heuristic function is initialized as an empty dictionary.

3.2 Computing Updated Heuristic Values

Following Algorithm 1, we now describe how updated heuristic values are obtained. While we can use Equation 1, as is done with deep reinforcement learning, the ILP system we are using, Popper, only returns hypotheses that entail all positive examples and do not entail any negative examples. Therefore, it is possible that, for a particular cost-to-go, we fail to learn a logic program. In this scenario, we would then fail to learn logic programs for any larger cost-to-go values since Equation 1 is simply a one-step lookahead. To overcome this, we use A* search as our update method, which is effectively a multi-step lookahead. Given the generated states, we run an A* search with the current learned heuristic function for a fixed number of iterations. For the states where a path is found, we set their updated heuristic value to be that of the path cost found. We remove states where a path cost is not found from the dataset[3].

[3] An alternative approach would be to use the nodes expanded so far to compute a lower bound, which we will explore in future work.

Algorithm 1 Heuristic Training

Input: initial heuristic function h, number of iterations I
for i in $[0, I)$ **do**
  $\mathcal{D} \leftarrow \{\}$
  $\mathcal{S}_i \leftarrow generate\_states()$
  for $s \in \mathcal{S}_i$ **do**
    $h' \leftarrow$ updated_heur_val(s, h)
    $\mathcal{D} \leftarrow \mathcal{D} \cup (s, h')$
  $h \leftarrow$ update_heur_func$(h, \mathcal{D})$
return h

Algorithm 2 Update Logic Heuristic

Input: dataset $\mathcal{D}$, target cost-to-go c
$E^+ = \{\}$
$E^- = \{\}$
for $(s, h') \in \mathcal{D}$ **do**
  if $h' >= c$ **then**
    $E^+ \leftarrow E^+ \cup \{s\}$
  else
    $E^- \leftarrow E^- \cup \{s\}$
$h[c] \leftarrow ILP(E^+, E^-)$
return h

3.3 Heuristic Function Update

To update the heuristic function, we will add another entry to the heuristic function dictionary. The key will be the smallest heuristic value, h', amongst the updated heuristic values such that h' is greater than all other heuristic value keys currently in the dictionary. We will refer to this as our target cost-to-go. We then generate positive and negative examples by comparing the updated heuristic values to the target cost-to-go. For each state, if its updated heuristic value is greater than or equal to the target cost-to-go, it is a positive example; otherwise, it is a negative example. We then use Popper to learn a logic program for the target cost-to-go. The logic program heuristic function update is outlined in Algorithm 2, source code is provided at: `https://github.com/Rojina99/HeurSearchILP`.

Predicate Reuse We can build on the given background knowledge by allowing predicates learned from a lower cost-to-go value to be used to learn logic programs for a higher cost-to-go value. In our approach, when learning a logic program in Algorithm 2, we reuse each clause learned for the logic program for the previous target cost-to-go by renaming each clause and adding it to the background knowledge and hypothesis space. We view this process as a kind of predicate invention that builds on the dynamic programming concept of solving problems by reusing solutions to subproblems.

4 Experiments

We test our algorithm to learn heuristic functions for logic programs on the 8-puzzle, which is a 3×3 grid sliding tile puzzle often used for validating pathfinding algorithms. Since the state space of this puzzle is relatively small (181,440 states), we obtain the true cost-to-go for all states and compare this to the learned heuristic values. We then test how the learned heuristic function performs when used with A* search. Finally, we test the performance of a heuristic function learned using supervised learning on all states in the state space. Our

background file contains basic information about the 8-puzzle, such as tiles, rows, and columns. It contains relations between these entities, such as adjacency and a tile being on a particular row on in its correct place. It also contains predicates that define when a row or column is complete. More detail can be found in Appendix 8.

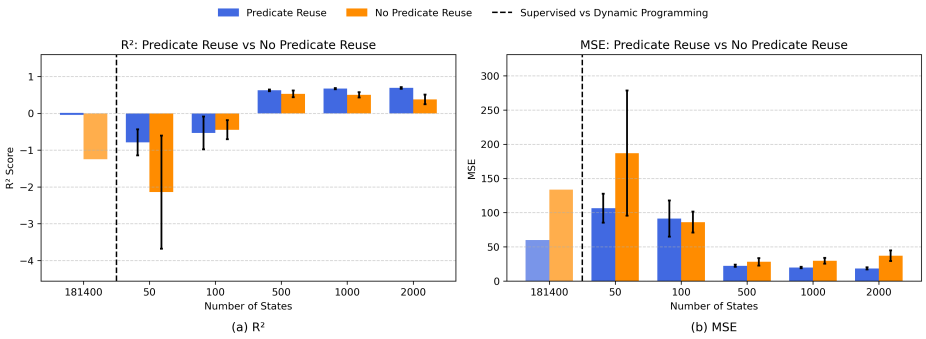

Fig. 1. (a)R^2 score and (b) Mean Squared Error for supervised learning (popper time limit 1 hr) and their mean and standard deviation for the dynamic programming (popper time limit 1200s) case across 5 sample sizes with and without predicate reuse.

Table 1. Heuristic search with the learned heuristics; values only for solved states.

States per cost-to-go	Predicate Reuse	Len	Nodes	Secs	Nodes/Sec	Solved (%)	Optimal (%)
181440	Yes	17.27	118.33	155.58	1.02	90.0	100.0
181440	No	17.73	1332.29	132.59	10.21	98.0	100.0
States per cost-to-go	**Predicate Reuse**	**Len (Mean ± SD)**	**Nodes (Mean ± SD)**	**Secs (Mean ± SD)**	**Nodes/Sec (Mean ± SD)**	**Solved (%) (Mean ± SD)**	**Optimal (%) (Mean ± SD)**
1000	Yes	18.96 ± 0.33	166.53 ± 7.93	86.06 ± 4.34	2.65 ± 0.18	99.60 ± 0.89	64.30 ± 6.67
1000	No	19.18 ± 0.29	250.16 ± 48.21	53.39 ± 10.54	5.30 ± 0.49	100.0	62.40 ± 9.63
2000	Yes	18.38 ± 0.05	136.39 ± 12.61	117.93 ± 10.40	1.57 ± 0.13	98.40 ± 1.67	78.40 ± 5.75
2000	No	18.38 ± 0.12	234.04 ± 66.96	69.62 ± 12.97	3.74 ± 0.52	100.0	80.80 ± 4.60

While training the heuristic function, we conducted five experiments for both predicate reuse and no predicate reuse. For each scenario, we ran experiments for 5 values for the number of generated states, ranging from 50 to 2000. Since the states are generated randomly, the results could be different across runs; therefore, we ran each experiment five times, resulting in a total of 50 experiments. We provide a timeout of 4 hours to A* while updating the heuristic values and also run A* for a maximum of 1000 iterations. We also include a comparison to a supervised learning case where we obtain the true cost-to-go for all 181,440 states and run Algorithm 2 to train the heuristic function with all possible target cost-to-go values and a time limit of 1 hour for learning rules with Popper. During testing, we test the accuracy of the heuristic function at predicting the true

cost-to-go using 942 states with cost-to-go values ranging from 1 to 31. In Figure 1, we show the mean and standard deviation of both R^2 and Mean Squared Error(MSE) between the predicted and ground-truth cost-to-go on this test set. We also evaluate the performance of the learned heuristic function combined with A* search to solve 50 test states. We run A* search for 10,000 iterations with a time limit of 1,000 seconds. The results of this are shown in Table 1.

5 Discussion and Future Work

Figure 1 shows that predicate reuse results in a more accurate heuristic function, as measured by R^2 and mean squared error, in almost every case. Furthermore, increasing the number of states generated with reinforcement learning tends to lead to a more accurate heuristic function. However, we see that in the supervised learning case, the heuristic function is less accurate. We believe this is due to the fact that, since we are using the entire dataset, there are more negative examples and, therefore, Popper requires more time to find a hypothesis that does not entail any negative examples. In fact, when increasing the time limit of the supervised learning case from 1 hour to 120 hours, the accuracy of the heuristic function improves (see Appendix). Future work can use ILP methods that attempt to find the most accurate hypothesis while allowing for misclassifications, such as those that learn from noisy data [22, 11]. We also observe that with sample sizes of 50 and 100, the heuristic function can accurately predict only smaller cost-to-go values. This may be because there are not enough positive samples at higher cost-to-go values. Once the sample size is increased, we can observe that the heuristic function can also correctly predict larger cost-to-go (see Appendix for plots).

Table 1 shows that predicate reuse results in the search algorithm expanding fewer nodes, which is expected for more accurate heuristic functions. However, the number of nodes processed per second and the time required to solve problems increases for predicate reuse. As a result, predicate reuse solves fewer problems. This may be because the heuristic function is slower when reusing predicates from prior target cost-to-go values. Furthermore, DNN heuristic functions often expand more nodes per second by several orders of magnitude [1] because they exploit graphics processing units for parallelism. Future work can look for ways to speed up the application of logic programs by exploiting parallelism.

6 Related Work

The method of bias reformulation is explored in the paper [15] to build reusable abstractions across tasks by permanently adding the learned predicates to the hypothesis space. In contrast, the predicate reuse reformulates bias more indirectly, where clauses learned at lower depths are added to background knowledge. Similarly, [21] learns heuristic functions in an adversarial setting in a domain with a two-player game, Shogi, where a player is trying to win against an active opponent, but they first learn local move comparisons, then convert them into

global heuristics. They also include restrictions for the game so that a player has to continue to check the opponent's King in the game, which is not easily transferable to other search problems without similar properties. In contrast, we formulate learning heuristics in terms of cost-to-go that is not dependent on any unique property of the 8-puzzle and can be transferable to any pathfinding domain where the domain can be expressed in logical predicates.

Using ILP to learn robot strategies is discussed in [9], where strategies are represented as recursive logic programs corresponding to sets of possible plans. The paper mainly focuses on finding efficient strategies with program learning entirely based on logic programs rather than search algorithms. Our paper focuses on learning heuristics for pathfinding problems that are represented as logic programs.

7 Conclusion

We present a novel algorithm for learning domain-specific heuristic functions represented as logic programs. We also introduce a method to reuse predicates to obtain more accurate heuristic functions. This work introduces new possibilities for learning explainable heuristic functions and discovering new knowledge from pathfinding problems.

8 Appendix

8.1 Appendix 1: Binary Search over Logic Program

We use binary search to obtain the heuristic value for the input state. Algorithm 3 provides an overview of the underlying working principle to obtain heuristic value. Given a state as input, we search for the logic program with maximum cost-to-go that entails the state. Logic program heuristic function h is represented as a dictionary to map the logic programs with cost-to-go values. $ctgs\_sorted$ are the list of cost-to-go values for which the logic program is already learned, initially empty. Similarly, $h[c]$ refers to the logic program learned for the cost-to-go of c, and B is the background file updated for each cost-to-go. Source code is provided at: `https://github.com/Rojina99/HeurSearchILP`.

8.2 Appendix 2: Background File

The provided background file defines the predicates to model the 8-puzzle domain for ILP. We define `tile/1` predicates to represent all the tiles in the puzzle, where `tile(b)` denotes the blank tile, and `tile(t1)` through `tile(t8)` corresponds to tiles numbered 1 through 8, respectively. These are mapped to numeric identifiers using predicates like `tile0(b)`, `tile1(t1)` up to `tile8(t8)` for convenience in indexing. The 8-puzzle grid is represented using the `indx/1`, which ranges from `idx1` to `idx9`, corresponding to the nine positions in the 3×3 grid, indexed

Algorithm 3 $h(s)$ via Binary Search over Logic Program

$low \leftarrow 0$
$ctgs\_sorted \leftarrow \text{keys}(h)$
$high \leftarrow |ctgs\_sorted| - 1$
$best \leftarrow \text{None}$
while $low \leq high$ **do**
  $mid \leftarrow \lfloor (low + high)/2 \rfloor$
  $c \leftarrow ctgs\_sorted[mid]$
  if $h[c] \cup B \models s$ **then**
    $best \leftarrow c$
    $low \leftarrow mid + 1$
  else
    $high \leftarrow mid - 1$
if $best \neq \text{None}$ **then**
  return $best$
else
  return 0

row-wise from left to right and top to bottom. For example, `idx1` is the top-left corner, and `idx9` is the bottom-right. To capture relationships among the indices, we define predicates such as `beforeto(I1, I2)`, which represents the information that index `I1` appears before `I2` in the left-to-right reading of the flattened puzzle grid, `adjacent_horiz(I1, I2)` for indices that are horizontally adjacent, `above(I1, I2)` to represent index `I1` is directly above `I2` in the grid. To locate tiles within a puzzle state (represented as a 9-element list), we use the `onrow(State, Tile, Index)` predicate to identify the index where a given tile appears. We define `inplace_clause(State, Tile)` to denote that a tile is in its correct position and `not_inplace_clause(State, Tile)` to indicate it is not. We also define predicates to check whether rows and columns are completed. The predicates `row1_comp/1`, `row2_comp/1`, and `row3_comp/1` verify that all tiles in the first, second, and third rows are correctly placed. Also, `col1_comp/1`, `col2_comp/1`, `col3_comp/1` verify column-wise correctness. The background file we use to describe the 8-puzzle domain is outlined below.

```
% background file starts

tile(b).
tile(t1).
tile(t2).
tile(t3).
tile(t4).
tile(t5).
tile(t6).
tile(t7).
tile(t8).

tile0(b).
```

```
tile1(t1).
tile2(t2).
tile3(t3).
tile4(t4).
tile5(t5).
tile6(t6).
tile7(t7).
tile8(t8).

indx(idx1).
indx(idx2).
indx(idx3).
indx(idx4).
indx(idx5).
indx(idx6).
indx(idx7).
indx(idx8).
indx(idx9).

indx1(idx1).
indx2(idx2).
indx3(idx3).
indx4(idx4).
indx5(idx5).
indx6(idx6).
indx7(idx7).
indx8(idx8).
indx9(idx9).

beforeto(idx1, idx2).
beforeto(idx2, idx3).
beforeto(idx3, idx4).
beforeto(idx4, idx5).
beforeto(idx5, idx6).
beforeto(idx6, idx7).
beforeto(idx7, idx8).
beforeto(idx8, idx9).

adjacent_horiz(idx1, idx2).
adjacent_horiz(idx2, idx3).
adjacent_horiz(idx4, idx5).
adjacent_horiz(idx5, idx6).
adjacent_horiz(idx7, idx8).
adjacent_horiz(idx8, idx9).

nextto_horiz(I1, I2) :- adjacent_horiz(I1, I2).
nextto_horiz(I1, I2) :- adjacent_horiz(I2, I1).

above(idx1, idx4).
above(idx2, idx5).
```

```
above(idx3, idx6).
above(idx4, idx7).
above(idx5, idx8).
above(idx6, idx9).

nextto_vert(I1, I2) :- above(I1, I2).
nextto_vert(I1, I2) :- above(I2, I1).

nextto(I1, I2) :- nextto_horiz(I1, I2).
nextto(I1, I2) :- nextto_vert(I1, I2).

onrow([Tile,_,_,_,_,_,_,_,_],Tile, idx1).
onrow([_,Tile,_,_,_,_,_,_,_],Tile, idx2).
onrow([_,_,Tile,_,_,_,_,_,_],Tile, idx3).
onrow([_,_,_,Tile,_,_,_,_,_],Tile, idx4).
onrow([_,_,_,_,Tile,_,_,_,_],Tile, idx5).
onrow([_,_,_,_,_,Tile,_,_,_],Tile, idx6).
onrow([_,_,_,_,_,_,Tile,_,_],Tile, idx7).
onrow([_,_,_,_,_,_,_,Tile,_],Tile, idx8).
onrow([_,_,_,_,_,_,_,_,Tile],Tile, idx9).

valid_var(T):- tile(T).

after_tile(t1, t2).
after_tile(t2, t3).
after_tile(t3, t4).
after_tile(t4, t5).
after_tile(t5, t6).
after_tile(t6, t7).
after_tile(t7, t8).

last_tile(t8).

goal([b, t1, t2, t3, t4, t5, t6, t7, t8]).

goal_index(Tile, GoalIndex) :-
    goal(GoalState),
    onrow(GoalState, Tile, GoalIndex).

inplace_clause(S, T) :-
    goal_index(T, GoalIndex),
    onrow(S, T, GoalIndex).

not_inplace_clause(S, T) :-
    goal_index(T, I_goal),
    onrow(S, T, I_current),
    distinct_indices(I_current, I_goal).

is_distinct(idx1, idx2).
is_distinct(idx1, idx3).
```

```
is_distinct(idx1, idx4).
is_distinct(idx1, idx5).
is_distinct(idx1, idx6).
is_distinct(idx1, idx7).
is_distinct(idx1, idx8).
is_distinct(idx1, idx9).
is_distinct(idx2, idx3).
is_distinct(idx2, idx4).
is_distinct(idx2, idx5).
is_distinct(idx2, idx6).
is_distinct(idx2, idx7).
is_distinct(idx2, idx8).
is_distinct(idx2, idx9).
is_distinct(idx3, idx4).
is_distinct(idx3, idx5).
is_distinct(idx3, idx6).
is_distinct(idx3, idx7).
is_distinct(idx3, idx8).
is_distinct(idx3, idx9).
is_distinct(idx4, idx5).
is_distinct(idx4, idx6).
is_distinct(idx4, idx7).
is_distinct(idx4, idx8).
is_distinct(idx4, idx9).
is_distinct(idx5, idx6).
is_distinct(idx5, idx7).
is_distinct(idx5, idx8).
is_distinct(idx5, idx9).
is_distinct(idx6, idx7).
is_distinct(idx6, idx8).
is_distinct(idx6, idx9).
is_distinct(idx7, idx8).
is_distinct(idx7, idx9).
is_distinct(idx8, idx9).

distinct_indices(I1, I2) :- is_distinct(I1, I2).
distinct_indices(I1, I2) :- is_distinct(I2, I1).

inplace_from(S, T) :-
    last_tile(T),
    inplace_clause(S, T).

inplace_from(S, T) :-
    \+ last_tile(T),
    after_tile(T, T_next),
    inplace_clause(S, T),
    inplace_from(S, T_next).

row1_comp(S) :- inplace_clause(S, b), inplace_clause(S, t1),
     inplace_clause(S, t2).
```

```
row2_comp(S) :- inplace_clause(S, t3), inplace_clause(S, t4),
    inplace_clause(S, t5).
row3_comp(S) :- inplace_clause(S, t6), inplace_clause(S, t7),
    inplace_clause(S, t8).

col1_comp(S) :- inplace_clause(S, b), inplace_clause(S, t3),
    inplace_clause(S, t6).
col2_comp(S) :- inplace_clause(S, t1), inplace_clause(S, t4),
    inplace_clause(S, t7).
col3_comp(S) :- inplace_clause(S, t2), inplace_clause(S, t5),
    inplace_clause(S, t8).

%background file ends
```

8.3 Appendix 3: Comparative Analysis of Learned Heuristic Function

In Figure 2, we present the median R^2 values computed over 942 test state samples, using heuristic function learned across five different runs for each sample state size(i.e., 50, 100, 500, 1000, 2000). The experiments were conducted under two conditions: with and without predicate reuse, resulting in a total of 10 plots.

From Figure 2(a), we can see the trend that as sample dataset size increases, median R^2 is increased, and the median Mean Squared Error(MSE) is decreased. Figure 2(b) also shows a similar trend up to a sample size of 500. However, the median R^2 drops after state size 1000, and the median MSE increases. This might be due to the reason that as the sample size increases, the number of negative examples is likely to increase. Since Popper tries to learn rules that do not entail negative examples, the provided time limit of 1200 seconds might not be sufficient to learn rules that do not entail negative examples. We can also see in Figure 2 that for state size 50, the heuristic function can predict some of the cost-to-go correctly up to 16 for predicate reuse and 15 for no predicate reuse, whereas, in the case of size 2000, it goes up to 26 for predicate reuse and up to 21 for no predicate reuse. Similarly, for sizes 100, 500, and 1000, the values increase to 18, 26, and 25 for predicate reuse and to 18, 23, and 22 for no predicate reuse, respectively. We can observe that the range of values for accurate prediction of cost-to-go does not increase monotonically. However, in the case of predicate reuse, Median R^2 shows a monotonic increment order. From this, we can conclude that even though the maximum range of correct prediction for cost-to-go is higher for state size 500, the heuristic learned with sample size 1000 predicts more cost-to-go correctly. The earlier cut-off on correctly predicted cost-to-go for a smaller sample size might be because there are not enough positive samples at a higher cost-to-go for the heuristic function to learn effective rules. Once the sample size is increased, we can observe that the heuristic function can also predict larger cost-to-go correctly.

We also learn the logic program rules and, thus, the heuristic function with full data samples. The result of testing this with 942 sample states is shown in

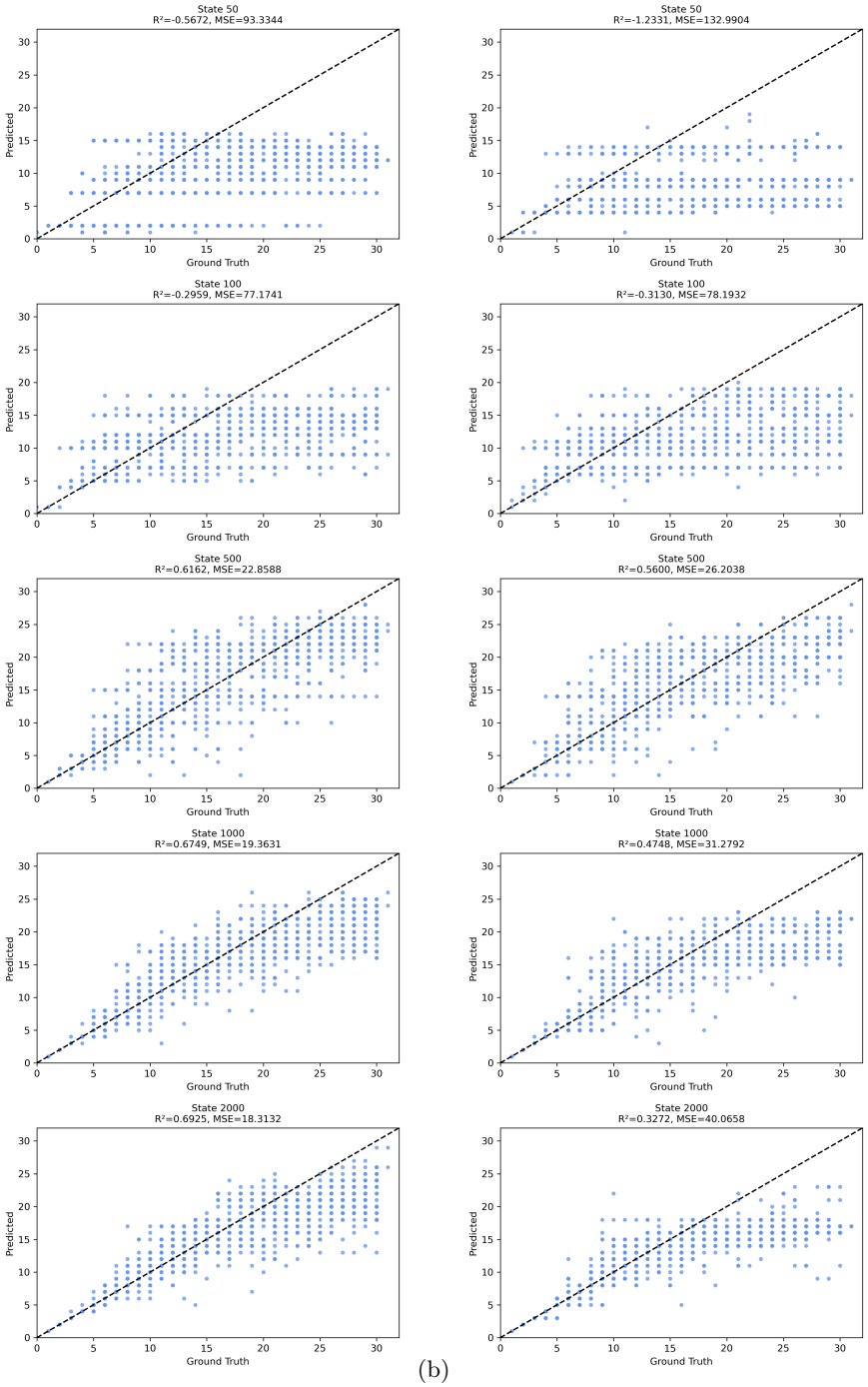

Fig. 2. (a) Median of R^2 score and Mean Square Error against five runs each for sample state size 50, 100, 500, 1000, and 2000 with predicate reuse (b) without predicate reuse.

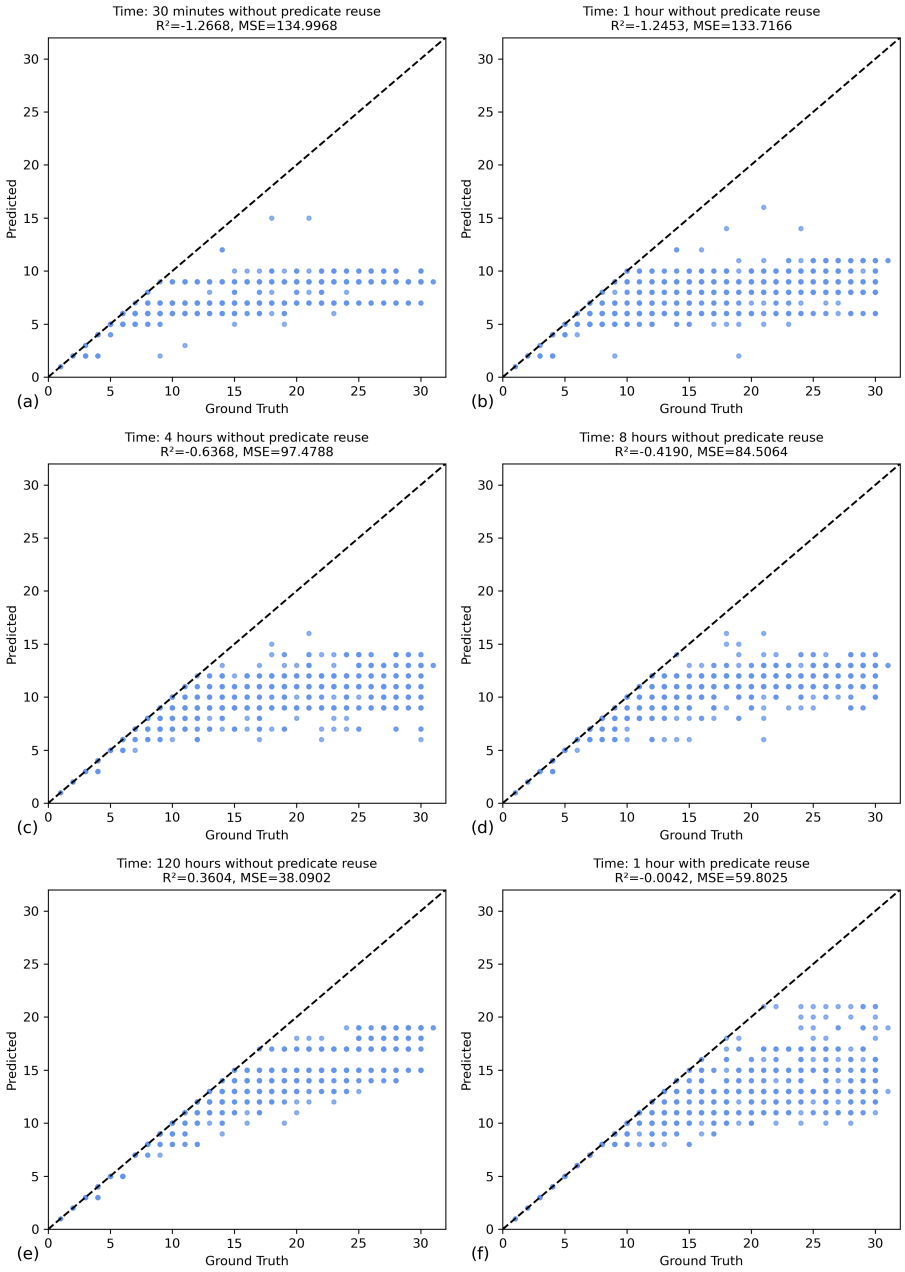

Fig. 3. (a,b,c,d,e) R^2 score and Mean Square Error for all dataset without predicate reuse for time limit of half an hour,1 hour,4 hours,8 hours and 120 hours, (f) with predicate reuse for limit of 1 hour.

Table 2. Comparative analysis of learned heuristics with all **181440** states, which is the total number of solvable states in 8-Puzzle with a time limit of 1 hour for predicate reuse and 30 minutes, 1 hour, 2 hours, 4 hours, 8 hours, and 120 hours without predicate reuse. Metrics reported include R^2 score and Mean Squared Error (MSE) between the predicted and ground-truth cost-to-go on 942 test samples.

States	Predicate Reuse	Time limit(hour/s)	R^2 Score	MSE
181440	Yes	1	−0.004	59.803
181440	No	0.5	−1.267	134.997
181440	No	1	−1.245	133.717
181440	No	2	−0.712	101.986
181440	No	4	−0.637	97.479
181440	No	8	−0.419	84.506
181440	No	120	0.360	38.090

Table 3. Final heuristic performance summary for 50 test samples run with A* search with and without predicate reuse. The heuristic function is learned with all **181440** states. Metrics reported include the average path length (Len), the number of expanded nodes (Nodes), solve time in seconds (Secs), search speed (Nodes/Sec), the percentages of problems solved and optimally solved problems.

States (hour/s)	Predicate Reuse	Len	Nodes	Secs	Nodes/Sec	Solved (%)	Optimal (%)
181440 (1 hour)	Yes	17.27	118.33	155.58	1.02	90.0	100.0
181440 (0.5 hours)	No	17.73	1357.49	117.61	11.65	98.0	100.0
181440 (1 hours)	No	17.73	1332.29	132.59	10.21	98.0	100.0
181440 (2 hours)	No	17.73	560.18	97.54	6.00	98.0	100.0
181440 (4 hours)	No	17.73	499.61	111.39	4.53	98.0	100.0
181440 (8 hours)	No	17.73	347.76	91.18	3.89	98.0	100.0
181440 (120 hours)	No	17.92	112.28	90.43	1.33	100.0	100.0

Table 2. We experiment with no-predicate reuse, using time limits that range from 30 minutes to 120 hours. From this, we can see that R^2 increases as the time limit increases and MSE decreases correspondingly. Since there is no variation in sample data size, we do not take the mean of R^2 like in the case of sampled states. Additionally, we also show the results for predicate reuse learned with a time limit of 1 hour. Since the size of the background file increases with predicate reuse, resulting in approximately 6,000 new rules learned, we were unable to obtain results beyond a cost-to-go of 21. The R^2 for predicate reuse with a 1-hour time limit is greater than that of no predicate reuse with an 8-hour time limit, and the MSE is lesser than that with an 8-hour time limit. Results from Table 2 are also shown in Figure 3 with a time limit of 2 hours excluded. From Figure 3(e), we can see that for no predicate reuse, there are some correct predictions up to cost-to-go 17, whereas, From 3(f), we can see it can predict some samples up to cost-to-go 21 correctly.

Table 3 presents the results of 50 test samples run with A* search, utilizing a heuristic function learned from a full dataset under various time limits. We allocate a time limit of 1,000 seconds for each state while running the A* algorithm,

which is set to execute for a maximum of 10,000 iterations. We only take an average of solved instances into account and omit the ones that are not solved. From the table, we can observe that the solved percentage for a time limit of 120 without predicate reuse is the highest, whereas that of 1 hour with predicate reuse is the lowest. We see that the time taken to solve the problem is slightly higher with predicate reuse. Because it takes more time to solve the problem, and we are using a limit of 1000 seconds, the percentage of solved problems is lower than in the case of no predicate reuse. However, the percentage of optimal problems solved is 100% for all variations of heuristic functions.

8.4 Appendix 4: Learned Rules with Sample State of Size 100 and Predicate Reuse

We show logic program rules obtained by one of the heuristic functions learned with a sample state of size 100 to provide an idea of learned rules with predicate reuse. We chose this version because it shows predicate reuse for some of the cost-to-go and is comprehensible. Other rules learned for higher sample sizes are longer, with approximately 1100 new rules for sample sizes of 500 and higher, so they are omitted here. The depth here refers to the lower bound on the cost-to-go from goal states. In depth 1, we can see the rule learned is `clause_depth 1_rule_1(V0) :- tile(V1),not_inplace_clause(V0,V1)`, which states that if it is a tile and it is not in place, then we can have a cost-to-go value of at least one. We can also observe that the rules are learned up to depth 22 with Popper, with a time limit of 1200 seconds for each cost-to-go.

```
=======================================
Final Logic Program Rules Learned by Depth
=======================================

% Rules from depth 1
clause_depth_1_rule_1(V0) :- tile(V1),not_inplace_clause(V0,V1).

% Rules from depth 2
clause_depth_2_rule_1(V0) :- tile4(V1),not_inplace_clause(V0,V1).
clause_depth_2_rule_2(V0) :- tile1(V1),not_inplace_clause(V0,V1).

% Rules from depth 3
clause_depth_3_rule_1(V0) :- tile(V2),not_inplace_clause(V0,V2),
    after_tile(V2,V1),not_inplace_clause(V0,V1).

% Rules from depth 4
clause_depth_4_rule_1(V0) :- tile6(V1),not_inplace_clause(V0,V1).
clause_depth_4_rule_2(V0) :- tile5(V2),not_inplace_clause(V0,V2),tile1(V1
    ),not_inplace_clause(V0,V1).

% Rules from depth 5
clause_depth_5_rule_1(V0) :- tile1(V1),not_inplace_clause(V0,V1),
    clause_depth_4_rule_1(V0).
```

```
clause_depth_5_rule_2(V0) :- tile2(V2),not_inplace_clause(V0,V2),
    after_tile(V2,V1),not_inplace_clause(V0,V1).
clause_depth_5_rule_3(V0) :- tile(V2),after_tile(V2,V3),
    not_inplace_clause(V0,V2),indx8(V1),onrow(V0,V3,V1).

% Rules from depth 6
clause_depth_6_rule_1(V0) :- clause_depth_5_rule_1(V0).
clause_depth_6_rule_2(V0) :- indx8(V2),tile3(V1),onrow(V0,V1,V2).
clause_depth_6_rule_3(V0) :- indx3(V1),tile4(V2),onrow(V0,V2,V1).
clause_depth_6_rule_4(V0) :- tile3(V1),not_inplace_clause(V0,V1),tile8(V2
    ),not_inplace_clause(V0,V2).

% Rules from depth 7
clause_depth_7_rule_1(V0) :- clause_depth_6_rule_1(V0).
clause_depth_7_rule_2(V0) :- tile(V4),after_tile(V4,V1),after_tile(V1,V3)
    ,goal_index(V4,V2),onrow(V0,V3,V2).
clause_depth_7_rule_3(V0) :- indx3(V1),tile5(V3),indx(V2),onrow(V0,V3,V2)
    ,is_distinct(V2,V1).

% Rules from depth 8
clause_depth_8_rule_1(V0) :- clause_depth_7_rule_3(V0).
clause_depth_8_rule_2(V0) :- tile8(V1),not_inplace_clause(V0,V1).
clause_depth_8_rule_3(V0) :- tile1(V1),indx4(V2),onrow(V0,V1,V2).

% Rules from depth 9
clause_depth_9_rule_1(V0) :- indx5(V1),tile3(V2),onrow(V0,V2,V1).
clause_depth_9_rule_2(V0) :- tile(V2),after_tile(V2,V3),indx9(V1),onrow(
    V0,V2,V1).

% Rules from depth 10
clause_depth_10_rule_1(V0) :- col1_comp(V0).
clause_depth_10_rule_2(V0) :- tile5(V1),indx4(V2),onrow(V0,V1,V2).
clause_depth_10_rule_3(V0) :- tile8(V2),not_inplace_clause(V0,V2),tile0(
    V1),not_inplace_clause(V0,V1).

% Rules from depth 11
clause_depth_11_rule_1(V0) :- tile6(V2),indx(V1),onrow(V0,V2,V1),indx4(V3
    ),is_distinct(V1,V3).
clause_depth_11_rule_2(V0) :- tile2(V2),indx(V1),onrow(V0,V2,V1),indx4(V3
    ),nextto(V3,V1).
clause_depth_11_rule_3(V0) :- tile(V1),goal_index(V1,V2),after_tile(V1,V4
    ),after_tile(V4,V3),not_inplace_clause(V0,V4),onrow(V0,V3,V2).

% Rules from depth 12
clause_depth_12_rule_1(V0) :- row1_comp(V0).
clause_depth_12_rule_2(V0) :- indx8(V1),tile2(V2),onrow(V0,V2,V1).
clause_depth_12_rule_3(V0) :- indx9(V1),tile1(V2),onrow(V0,V2,V1).
clause_depth_12_rule_4(V0) :- indx4(V3),tile(V1),indx7(V2),onrow(V0,V1,V2
    ),goal_index(V1,V4),is_distinct(V4,V3).
```

```
clause_depth_12_rule_5(V0) :- indx2(V2),tile7(V3),tile(V1),onrow(V0,V1,V2
    ),goal_index(V1,V4),onrow(V0,V3,V4).

% Rules from depth 13
clause_depth_13_rule_1(V0) :- clause_depth_12_rule_2(V0).
clause_depth_13_rule_2(V0) :- indx6(V1),tile6(V2),onrow(V0,V2,V1).
clause_depth_13_rule_3(V0) :- tile8(V1),tile2(V2),not_inplace_clause(V0,
    V2),not_inplace_clause(V0,V1).
clause_depth_13_rule_4(V0) :- indx(V1),tile1(V3),onrow(V0,V3,V1),indx6(V2
    ),is_distinct(V2,V1).
clause_depth_13_rule_5(V0) :- tile8(V3),tile4(V2),not_inplace_clause(V0,
    V2),indx8(V1),onrow(V0,V3,V1).

% Rules from depth 14
clause_depth_14_rule_1(V0) :- clause_depth_13_rule_4(V0),tile5(V1),
    not_inplace_clause(V0,V1).
clause_depth_14_rule_2(V0) :- tile2(V1),indx6(V2),onrow(V0,V1,V2).
clause_depth_14_rule_3(V0) :- tile8(V2),not_inplace_clause(V0,V2),tile7(
    V1),inplace_clause(V0,V1).
clause_depth_14_rule_4(V0) :- tile7(V1),indx5(V3),onrow(V0,V1,V3),tile5(
    V2),not_inplace_clause(V0,V2).
clause_depth_14_rule_5(V0) :- tile(V4),after_tile(V4,V1),goal_index(V4,V2
    ),goal_index(V1,V3),onrow(V0,V4,V3),onrow(V0,V1,V2).

% Rules from depth 15
clause_depth_15_rule_1(V0) :- tile7(V1),indx3(V2),onrow(V0,V1,V2).
clause_depth_15_rule_2(V0) :- tile6(V1),indx5(V2),onrow(V0,V1,V2).
clause_depth_15_rule_3(V0) :- tile6(V1),tile0(V3),not_inplace_clause(V0,
    V1),indx5(V2),onrow(V0,V3,V2).
clause_depth_15_rule_4(V0) :- tile7(V3),tile2(V1),not_inplace_clause(V0,
    V1),indx4(V2),onrow(V0,V3,V2).
clause_depth_15_rule_5(V0) :- tile2(V4),tile(V3),indx5(V2),onrow(V0,V3,V2
    ),goal_index(V3,V1),onrow(V0,V4,V1).

% Rules from depth 16
clause_depth_16_rule_1(V0) :- clause_depth_15_rule_3(V0).
clause_depth_16_rule_2(V0) :- clause_depth_15_rule_4(V0).
clause_depth_16_rule_3(V0) :- tile5(V2),indx1(V1),onrow(V0,V2,V1).
clause_depth_16_rule_4(V0) :- tile6(V1),not_inplace_clause(V0,V1),tile3(
    V2),inplace_clause(V0,V2).
clause_depth_16_rule_5(V0) :- tile6(V3),inplace_clause(V0,V3),tile8(V1),
    indx2(V2),onrow(V0,V1,V2).
clause_depth_16_rule_6(V0) :- indx2(V2),tile8(V3),not_inplace_clause(V0,
    V3),tile3(V1),onrow(V0,V1,V2).
clause_depth_16_rule_7(V0) :- tile4(V3),not_inplace_clause(V0,V3),tile5(
    V1),indx2(V2),onrow(V0,V1,V2).

% Rules from depth 17
clause_depth_17_rule_1(V0) :- clause_depth_16_rule_4(V0),
    clause_depth_16_rule_7(V0).
```

```
clause_depth_17_rule_2(V0) :- tile8(V1),indx4(V2),onrow(V0,V1,V2).
clause_depth_17_rule_3(V0) :- indx6(V2),tile6(V1),onrow(V0,V1,V2).
clause_depth_17_rule_4(V0) :- tile0(V1),indx2(V3),tile6(V2),onrow(V0,V1,
    V3),not_inplace_clause(V0,V2).
clause_depth_17_rule_5(V0) :- tile5(V4),tile(V1),goal_index(V1,V2),onrow(
    V0,V4,V2),indx1(V3),onrow(V0,V1,V3).

% Rules from depth 18
clause_depth_18_rule_1(V0) :- clause_depth_17_rule_2(V0).
clause_depth_18_rule_2(V0) :- indx5(V2),tile1(V3),indx(V1),is_distinct(V2
    ,V1),onrow(V0,V3,V1).
clause_depth_18_rule_3(V0) :- indx1(V1),tile5(V3),onrow(V0,V3,V1),tile7(
    V2),not_inplace_clause(V0,V2).
clause_depth_18_rule_4(V0) :- indx8(V4),tile(V2),after_tile(V2,V3),onrow(
    V0,V3,V4),goal_index(V3,V1),onrow(V0,V2,V1).

% Rules from depth 19
clause_depth_19_rule_1(V0) :- indx4(V2),tile(V3),after_tile(V3,V1),onrow(
    V0,V1,V2),clause_depth_18_rule_2(V0).
clause_depth_19_rule_2(V0) :- indx9(V2),tile(V1),indx1(V3),onrow(V0,V1,V3
    ),after_tile(V1,V4),onrow(V0,V4,V2).

% Rules from depth 20
clause_depth_20_rule_1(V0) :- indx1(V1),tile7(V2),onrow(V0,V2,V1).
clause_depth_20_rule_2(V0) :- tile7(V1),not_inplace_clause(V0,V1),tile0(
    V2),inplace_clause(V0,V2).
clause_depth_20_rule_3(V0) :- tile5(V3),tile(V4),indx9(V1),goal_index(V4,
    V2),onrow(V0,V3,V2),onrow(V0,V4,V1).

% Rules from depth 21
clause_depth_21_rule_1(V0) :- tile2(V1),indx8(V2),onrow(V0,V1,V2).
clause_depth_21_rule_2(V0) :- tile4(V1),tile1(V3),indx9(V2),onrow(V0,V1,
    V2),not_inplace_clause(V0,V3).
clause_depth_21_rule_3(V0) :- indx2(V2),tile8(V3),onrow(V0,V3,V2),tile2(
    V1),not_inplace_clause(V0,V1).
clause_depth_21_rule_4(V0) :- indx9(V3),tile7(V4),onrow(V0,V4,V3),indx6(
    V1),tile2(V2),onrow(V0,V2,V1).

% Rules from depth 22
clause_depth_22_rule_1(V0) :- clause_depth_21_rule_2(V0).
clause_depth_22_rule_2(V0) :- tile5(V1),indx2(V2),onrow(V0,V1,V2).
```

References

1. Agostinelli, F., McAleer, S., Shmakov, A., Baldi, P.: Solving the Rubik's cube with deep reinforcement learning and search. Nature Machine Intelligence **1**(8), 356–363 (2019)

2. Agostinelli, F., Panta, R., Khandelwal, V.: Specifying goals to deep neural networks with answer set programming. In: 34th International Conference on Automated Planning and Scheduling (2024)
3. Ai, L., Langer, J., Muggleton, S.H., Schmid, U.: Explanatory machine learning for sequential human teaching. Machine Learning **112**(10), 3591–3632 (2023)
4. Bellman, R.: Dynamic Programming. Princeton University Press (1957)
5. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-dynamic programming. Athena Scientific (1996)
6. Chen, B., Li, C., Dai, H., Song, L.: Retro*: learning retrosynthetic planning with neural guided A* search. In: International Conference on Machine Learning. pp. 1608–1616. PMLR (2020)
7. Cropper, A., Dumančić, S.: Inductive logic programming at 30: a new introduction. Journal of Artificial Intelligence Research **74**, 765–850 (2022)
8. Cropper, A., Morel, R.: Learning programs by learning from failures. Machine Learning **110**(4), 801–856 (2021)
9. Cropper, A., Muggleton, S.: Learning efficient logical robot strategies involving composable objects. In: IJCAI. No. 3423, AAAI Press/International Joint Conferences on Artificial Intelligence (2015)
10. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968)
11. Hocquette, C., Niskanen, A., Järvisalo, M., Cropper, A.: Learning mdl logic programs from noisy data. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 38, pp. 10553–10561 (2024)
12. Jin, J., Kim, K.: 3D cube algorithm for the key generation method: Applying deep neural network learning-based. IEEE Access **8**, 33689–33702 (2020)
13. Korf, R., Taylor, L.: Finding optimal solutions to the 24-puzzle. In: Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996). pp. 1202–1207 (1996)
14. Korf, R.E.: Finding optimal solutions to rubik's cube using pattern databases. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence. pp. 700–705. AAAI'97/IAAI'97, AAAI Press (1997), http://dl.acm.org/citation.cfm?id=1867406.1867515
15. Lin, D., Dechter, E., Ellis, K., Tenenbaum, J.B., Muggleton, S.H.: Bias reformulation for one-shot function induction (2014)
16. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
17. Muggleton, S.: Inductive logic programming. New Generation Computing **8**(4), 295–318 (1991)
18. Muggleton, S.: Inductive logic programming. New generation computing **8**, 295–318 (1991)
19. Muggleton, S.: Inverse entailment and progol. New generation computing **13**, 245–286 (1995)
20. Muggleton, S.H., Schmid, U., Zeller, C., Tamaddoni-Nezhad, A., Besold, T.: Ultra-strong machine learning: comprehensibility of programs learned with ilp. Machine Learning **107**, 1119–1140 (2018)
21. Nakano, T., Inuzuka, N., Seki, H., Itoh, H.: Inducing shogi heuristics using inductive logic programming. In: International Conference on Inductive Logic Programming. pp. 155–164. Springer (1998)

22. Oblak, A., Bratko, I.: Learning from noisy data using a non-covering ilp algorithm. In: International conference on inductive logic programming. pp. 190–197. Springer (2010)
23. Panta, R., Tavakoli, M., Geils, C., Baldi, P., Agostinelli, F.: Finding reaction mechanism pathways with deep reinforcement learning and heuristic search. In: ICAPS Workshop on Bridging the Gap between AI Planning and Reinforcement Learning. ICAPS (2024)
24. Pohl, I.: Heuristic search viewed as path finding in a graph. Artificial intelligence **1**(3-4), 193–204 (1970)
25. Qiuhao, C., Du, Y., Zhao, Q., Jiao, Y., Lu, X., Wu, X.: Efficient and practical quantum compiler towards multi-qubit systems with deep reinforcement learning. Quantum Science and Technology (2024)
26. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural networks **61**, 85–117 (2015)
27. Siddique, P.J., Gue, K.R., Usher, J.S.: Puzzle-based parking. Transportation Research Part C: Emerging Technologies **127**, 103112 (2021)
28. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
29. Veronese, C., Meli, D., Bistaffa, F., Rodríguez-Sot, M., Farinelli, A., Rodríguez-Aguilar, J.A.: Inductive logic programming for transparent alignment with multiple moral values. vol. 3615, p. 84 – 88 (2023)