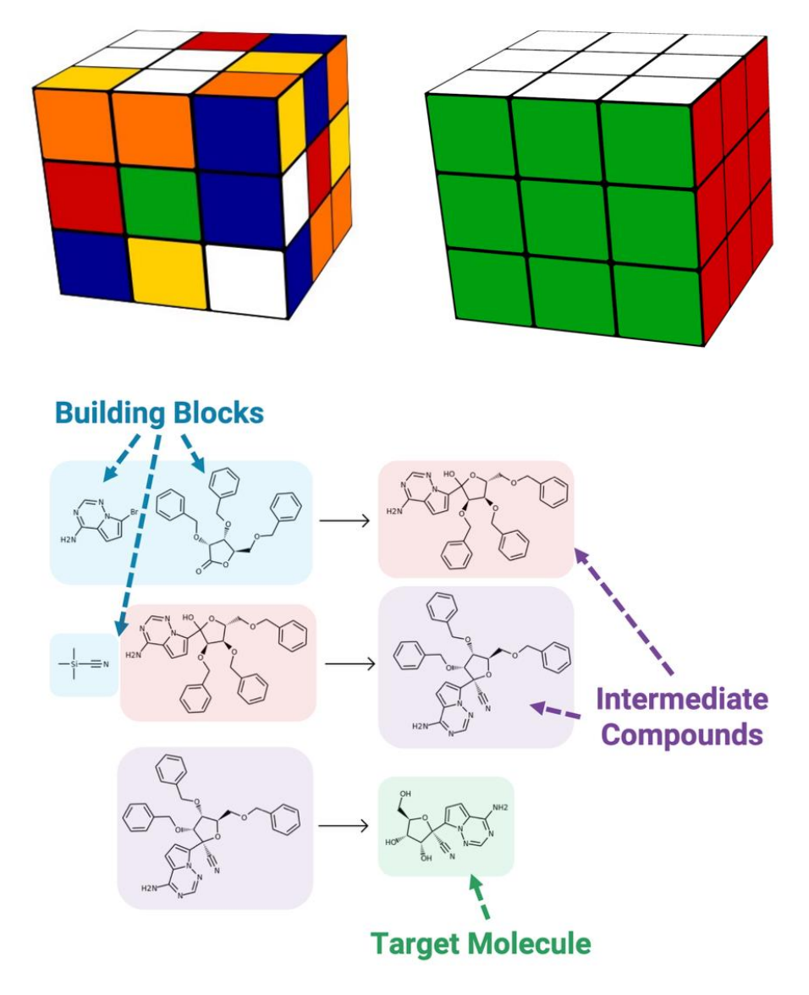
Inductive Logic Programming for Heuristic Search

Rojina Panta[1,2], Vedant Khandelwal[1,2],
Celeste Veronese[3,4], Amit Sheth[1,2], Daniele Meli[3,4], Forest Agostinelli[1,2]

1. AI Institute, University of South Carolina, Columbia, South Carolina, USA
2. Department of Computer Science and Engineering, University of South Carolina, Columbia, South Carolina, USA
3. Department of Computer Science, University of Verona, Italy
4. Intelligent Systems Lab, University of Verona, Italy

Pathfinding

- ❖ Pathfinding aims to find a sequence of actions that forms a path between a given start state and a given goal.
- ❖ Preference is given for minimum cost paths.
- ❖ Some examples of pathfinding problems are puzzle solving, chemical synthesis, and robotics.



Motivation

- ❖ Methods like Deep Reinforcement Learning (DRL) can help us learn strong heuristic functions, but neural networks are black boxes and hard to interpret.
- ❖ Logic programs are somewhat explainable, yet methods to learn heuristic functions represented as logic programs are missing.
- ❖ We present an algorithm to learn heuristic functions represented as logic programs using Dynamic Programming and Inductive Logic Programming (ILP).

Problem Statement

- ❖ Deep Neural Network (DNN) can directly map states to numbers, i.e., states to cost to go (ctg).
- ❖ Modern ILP systems, such as Popper, do classification.
- ❖ How can we use this to do regression instead so that it can approximate the true cost to go of some dataset?

{1: [rule1], 2: [rule2 ,rule3, rule4], 3:[rule5, rule6], 4: [rule7, rule 8, rule9], ……. }

Our Approach

Mapping States to Numbers

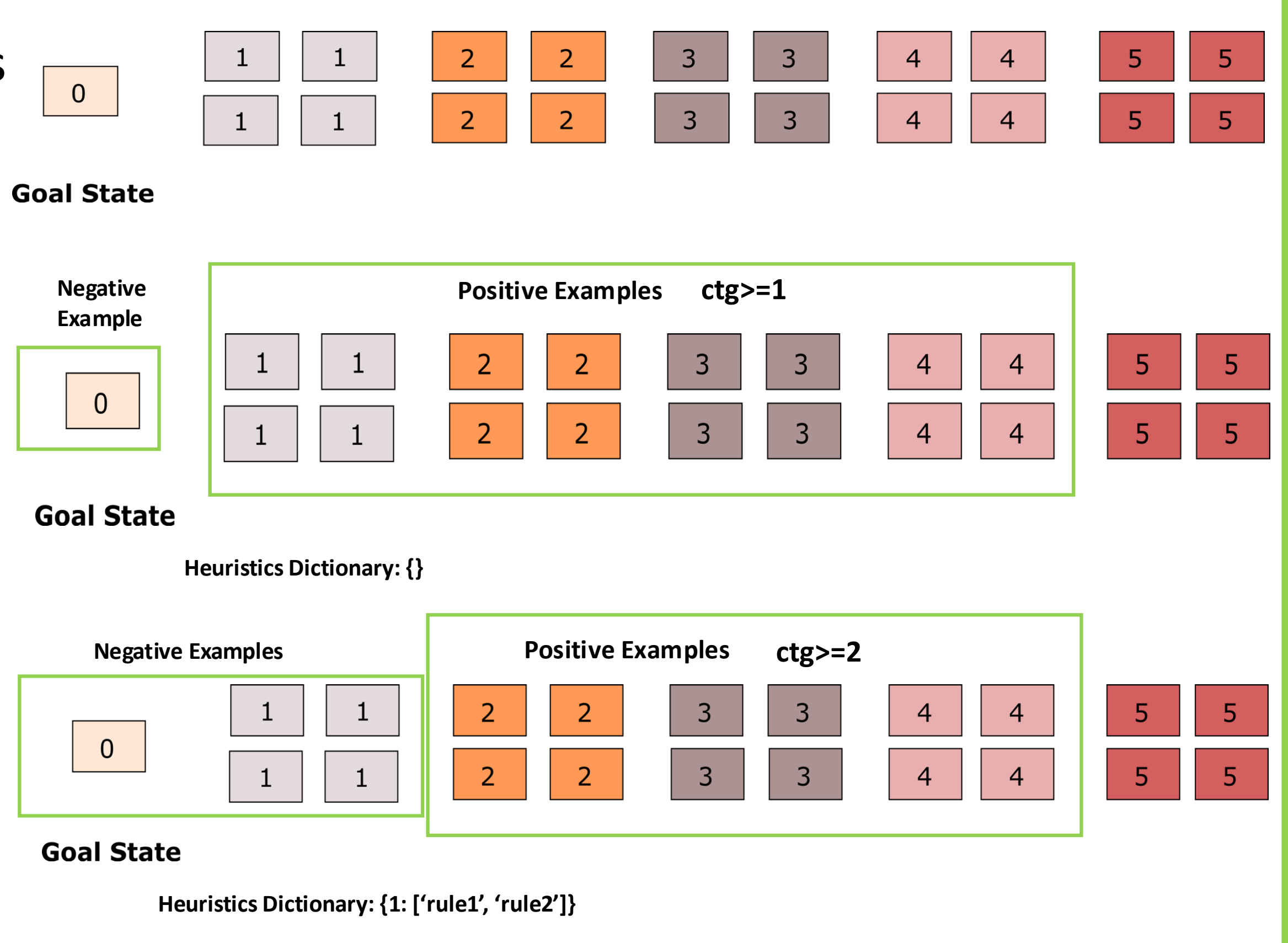
- ❖ If we have our heuristic dictionary with logic program rules, how can we map our states to the numbers?

Heuristics Dictionary: {1: [rule1], 2: [rule2 ,rule3, rule4], 3:[rule5, rule6], 4: [rule7, rule 8, rule9], …….}

- ❖ Keys represent the ctg.
- ❖ We check our state with rules at ctg>= 1 and see whether it satisfies rule 1. If not, return 0; otherwise, return 1. If it satisfies 1, we check 2. If it satisfies 2 we check for 3 and so on.

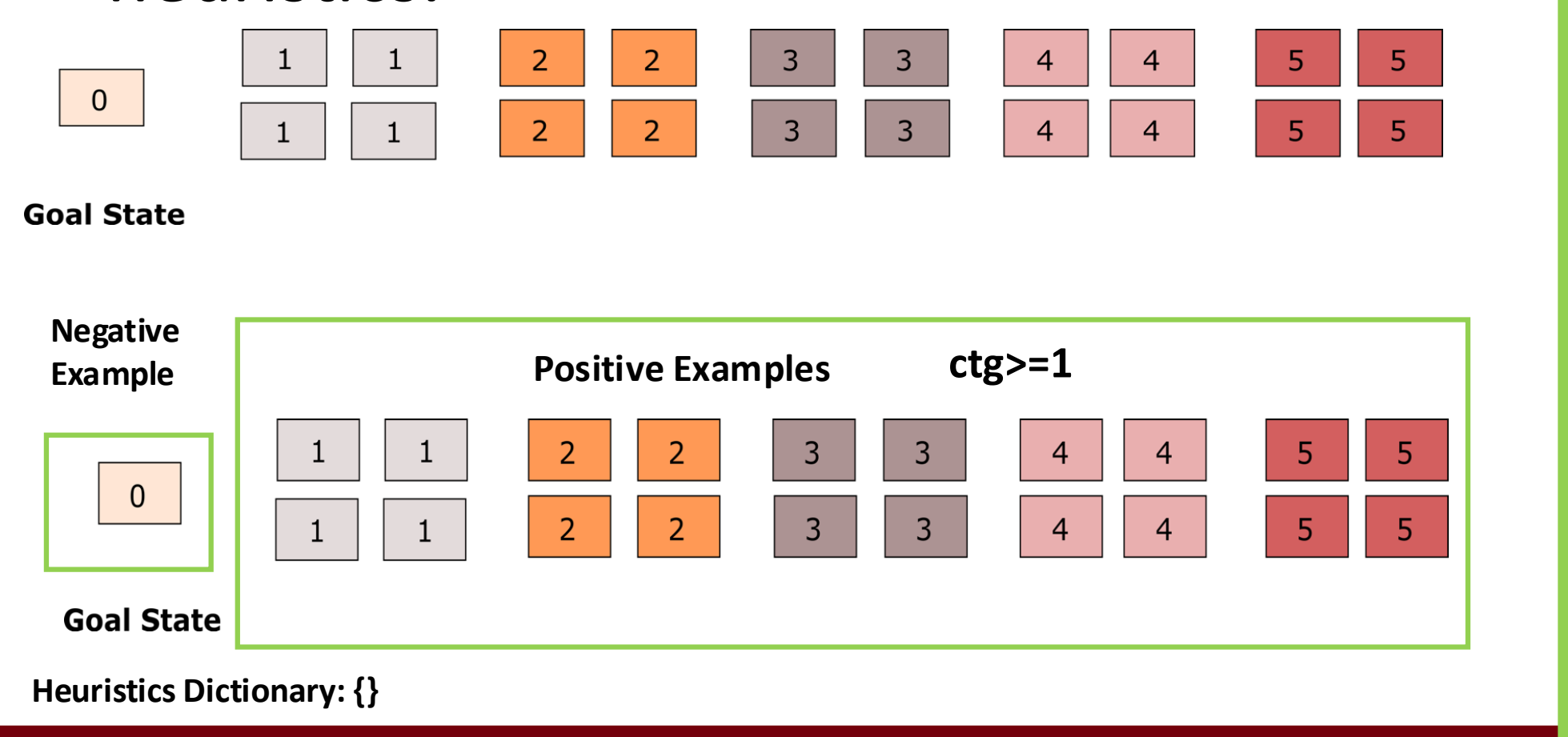
Learning Logic Program Heuristics

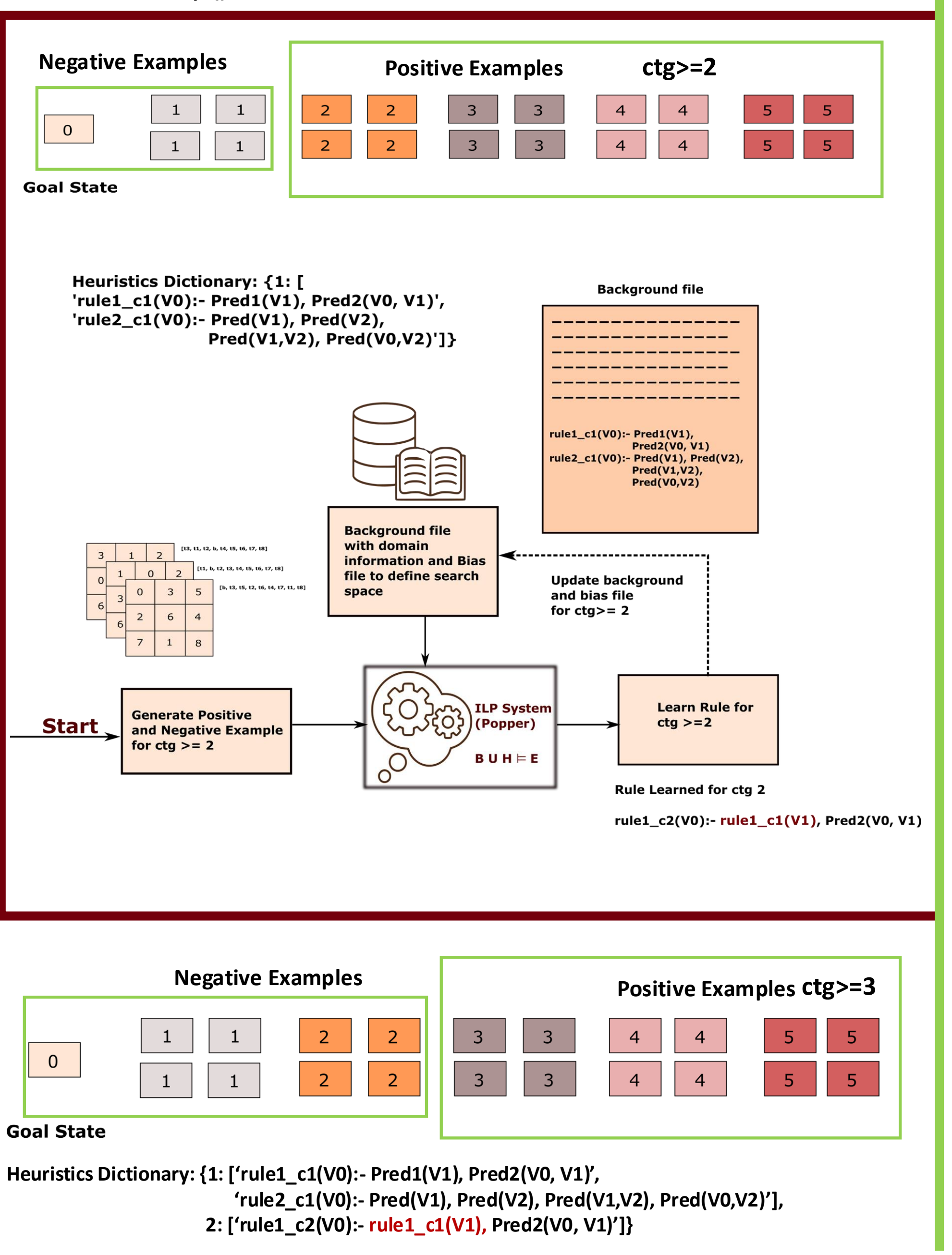
- ❖ We have a supervised dataset mapping states to ctg for the 8-puzzle domain.
- ❖ We need a method to learn a heuristic function.
- ❖ Start with an empty heuristic dictionary.
- ❖ State with ctg 0 is represented as a negative example, and everything else as a positive example.
- ❖ Pass it to the ILP system, Popper.
- ❖ Update heuristic function with the rule learned for ctg>=1.
- ❖ Update positive and negative examples.
- ❖ We use updated states to learn the rules for the ctg>=2.



Learning Logic Program Heuristics with Predicate Reuse

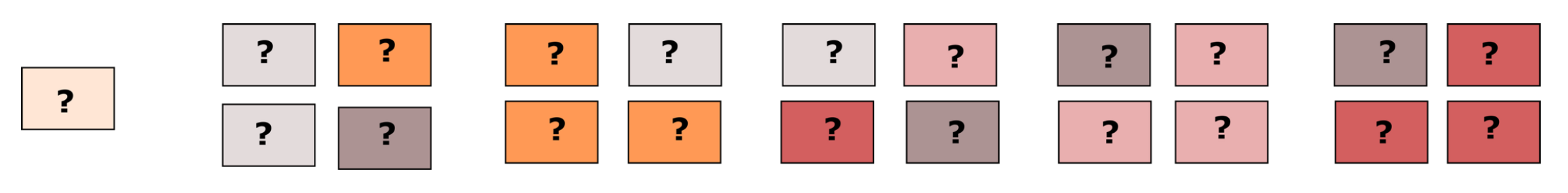
- ❖ Reuse clause learned for previous target ctg by renaming each clause and adding it to the background knowledge and hypothesis space when learning for larger values.
- ❖ Enables a dynamic programming style build up from simple to complex heuristics.



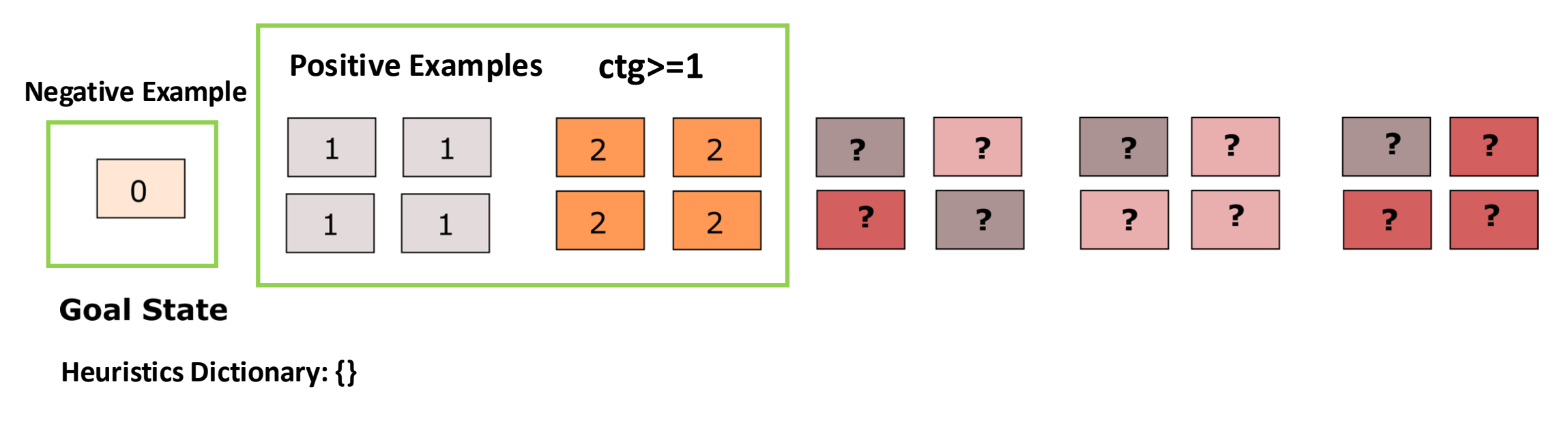

Negative Examples
Positive Examples ctg>=2
Goal State
Heuristics Dictionary: {1: [
'rule1_c1(V0):- Pred1(V1), Pred2(V0, V1)',
'rule2_c1(V0):- Pred(V1), Pred(V2), Pred(V1,V2), Pred(V0,V2)']}
Background file
rule1_c1(V0):- Pred1(V1), Pred2(V0, V1)
rule2_c1(V0):- Pred(V1), Pred(V2), Pred(V1,V2), Pred(V0,V2)
Background file with domain information and Bias file to define search space
Update background and bias file for ctg>= 2
Start
Generate Positive and Negative Example for ctg >= 2
ILP System (Popper)
B U H ⊨ E
Learn Rule for ctg >=2
Rule Learned for ctg 2
rule1_c2(V0):- rule1_c1(V1), Pred2(V0, V1)
Negative Examples
Positive Examples ctg>=3
Goal State
Heuristics Dictionary: {1: ['rule1_c1(V0):- Pred1(V1), Pred2(V0, V1)',
'rule2_c1(V0):- Pred(V1), Pred(V2), Pred(V1,V2), Pred(V0,V2)'],
2: ['rule1_c2(V0):- rule1_c1(V1), Pred2(V0, V1)']}

Learning Logic Program Heuristics with Search

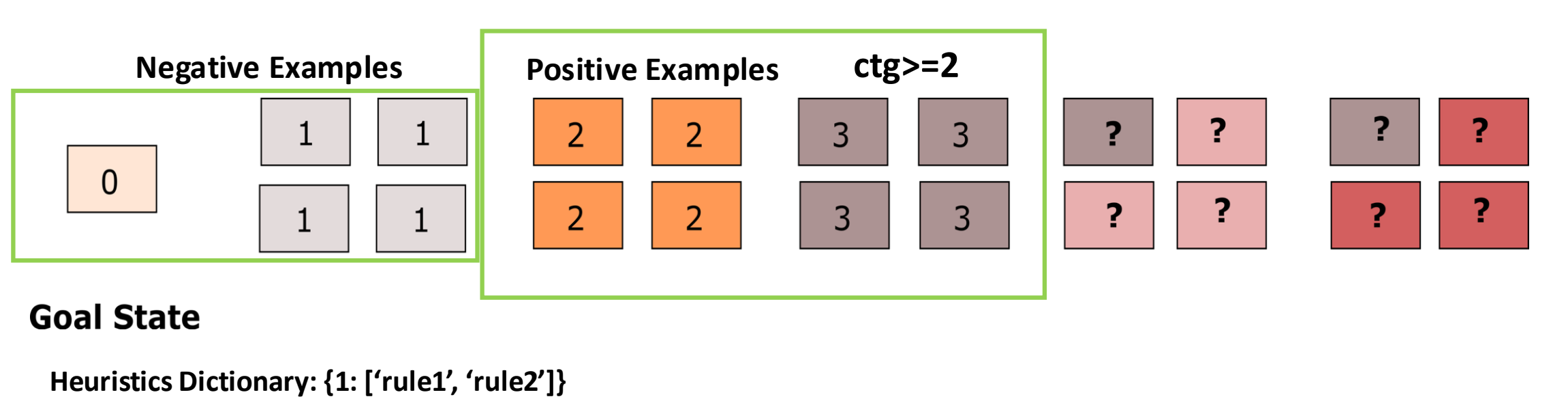
- ❖ What if we do not know the ctg values of any state?
- ❖ We can use a special case of a heuristic search algorithm, A* search, along with a heuristic function, to get the path cost of these states.



- ❖ A* runs for 1000 iterations.
- ❖ We first get the states with different path costs, classify states with path cost 0 as a negative example, classify all other states as positive, and pass them to the ILP system Popper.



- ❖ We update our heuristic function with the rule learned for the ctg>=1.
- ❖ We use the updated heuristic function to generate a dataset at ctg>=2.
- ❖ We classify states with path costs 0 and 1 as negative examples and all other states as positive, and pass them to the ILP system Popper.



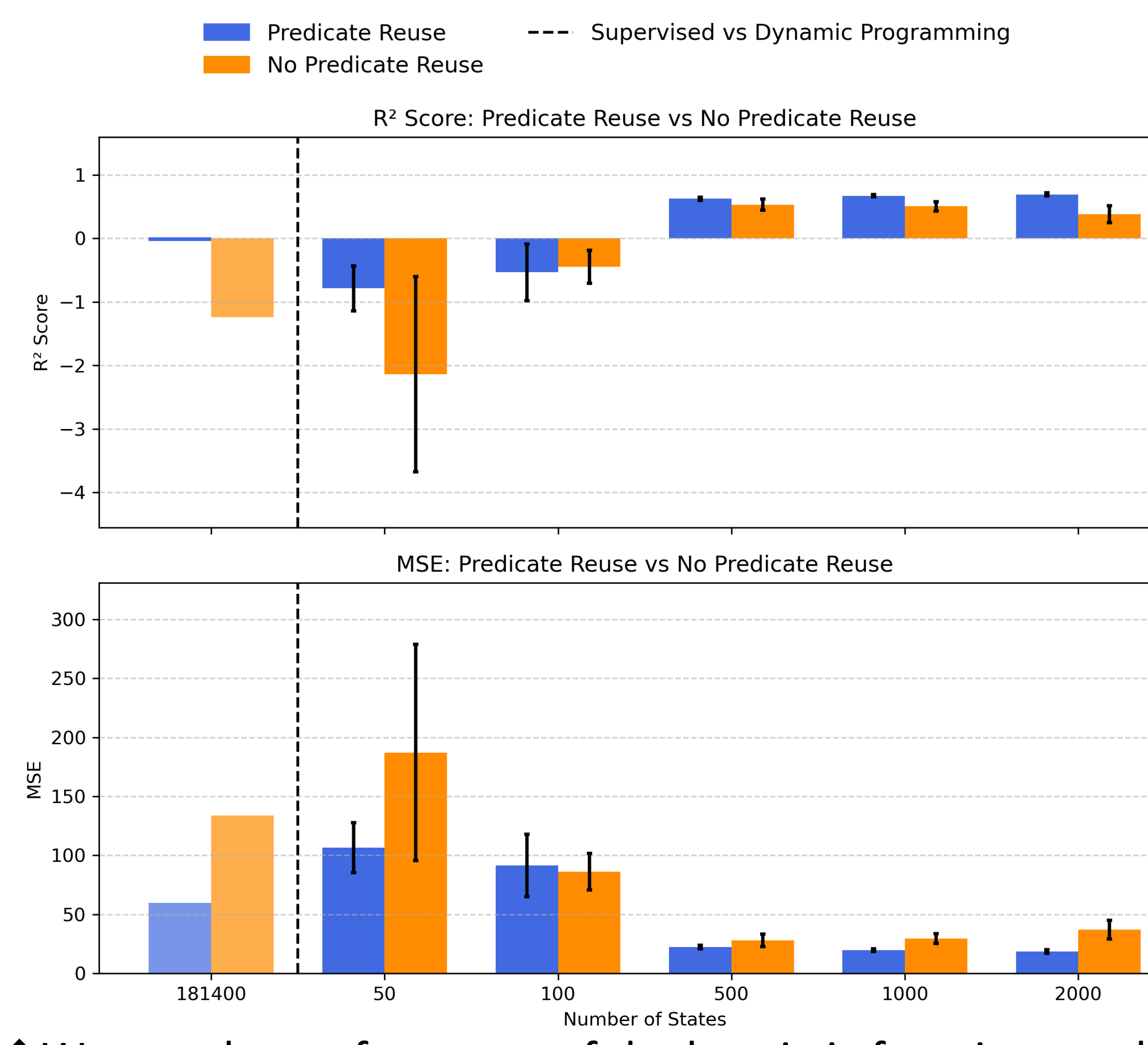
- ❖ We can also use predicate reuse with A* search, which allows the dynamic programming style build up from simple to complex heuristics.

```
-------------- ctg >=1 --------------
c1_r0_f(V0):- tile(V1),not_inplace_clause(V0,V1).
-------------- ctg >=2 --------------
c2_r0_f(V0):- tile6(V1),not_inplace_clause(V0,V1).
c2_r1_f(V0):- tile4(V1),not_inplace_clause(V0,V1).
c2_r2_f(V0):- tile2(V1),not_inplace_clause(V0,V1).
c2_r3_f(V0):- tile7(V1),not_inplace_clause(V0,V1).
c2_r4_f(V0):- tile5(V1),not_inplace_clause(V0,V1).
c2_r5_f(V0):- tile8(V1),not_inplace_clause(V0,V1).
```

```
-------------- ctg >=3 --------------
c3_r0_f(V0):- c2_r5_f(V0).
c3_r1_f(V0):- c2_r4_f(V0).
c3_r2_f(V0):- c2_r3_f(V0).
c3_r3_f(V0):- row2_comp(V0),c2_r0_f(V0).
c3_r4_f(V0):- c2_r1_f(V0),c2_r0_f(V0).
c3_r5_f(V0):- c2_r2_f(V0),c2_r1_f(V0).
c3_r6_f(V0):- c2_r2_f(V0),col2_comp(V0).
c3_r7_f(V0):- tile3(V2),not_inplace_clause(V0,V2),
                tile1(V1),not_inplace_clause(V0,V1).
```

Results

- ❖ We test the accuracy of the heuristic function at predicting the true ctg of 942 states with ctg values ranging from 1 to 31.
- ❖ The R^2 score and Mean Squared Error(MSE) results for supervised learning and dynamic programming datasets are shown.
- ❖ Dynamic programming with predicate reuse outperforms supervised learning for state size>=500.



- ❖ We test the performance of the heuristic function combined with an A* search on 50 test samples by running an A* search for 10,000 iterations with a time limit of 1,000 seconds.

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 ± 7.49	86.06 ± 4.39	2.65 ± 0.16	99.60 ± 0.89	64.30 ± 6.18
1000	No	19.18 ± 0.28	250.16 ± 43.51	53.40 ± 10.97	5.30 ± 0.47	100.0	62.40 ± 9.40
2000	Yes	18.38 ± 0.05	136.39 ± 11.39	117.93 ± 8.24	1.57 ± 0.14	98.40 ± 1.67	78.40 ± 5.17
2000	No	18.38 ± 0.12	234.04 ± 63.85	69.62 ± 12.89	3.74 ± 0.51	100.0	80.80 ± 4.09

Example of Learned Predicate

{tile(V2),after_tile(V2,V3),not_inplace_clause(V0,V3), goal_index(V3,V4),indx4(V1),distinct_indices(V4,V1)'}]:
"There exists a successor tile V3 (one of t2..t8) such that in state V0, V3 is out of place, and V3's correct goal position is not idx4."

3	1	2
4	0	5
6	7	8

- ❖ Since we can understand our heuristics more easily than DNN heuristics, this opens a new avenue for explainable pathfinding.

Future Work

- ❖ We observed that the supervised learning case had lower accuracy due to more negative examples. Future work can use ILP methods that attempt to find the more accurate hypothesis while allowing for misclassifications, such as those that learn from noisy data [1, 2].
- ❖ Compared to neural approaches [3], logic programs lack GPU-level parallelism, creating a performance gap for large-scale search. We can look for ways to speed up the application of logic programs by exploiting parallelism.

References

1. Hocquette, Céline, et al. "Learning MDL logic programs from noisy data." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 38. No. 9. 2024.
2. Oblak, Andrej, and Ivan Bratko. "Learning from noisy data using a non-covering ILP algorithm." International conference on inductive logic programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
3. Agostinelli, Forest, et al. "Solving the Rubik's cube with deep reinforcement learning and search." Nature Machine Intelligence 1.8 (2019): 356-363.
4. https://cse.sc.edu/~foresta/assets/files/SoCSMasterClass.pdf. (Accessed, 2025)