Learning programs by learning from failures

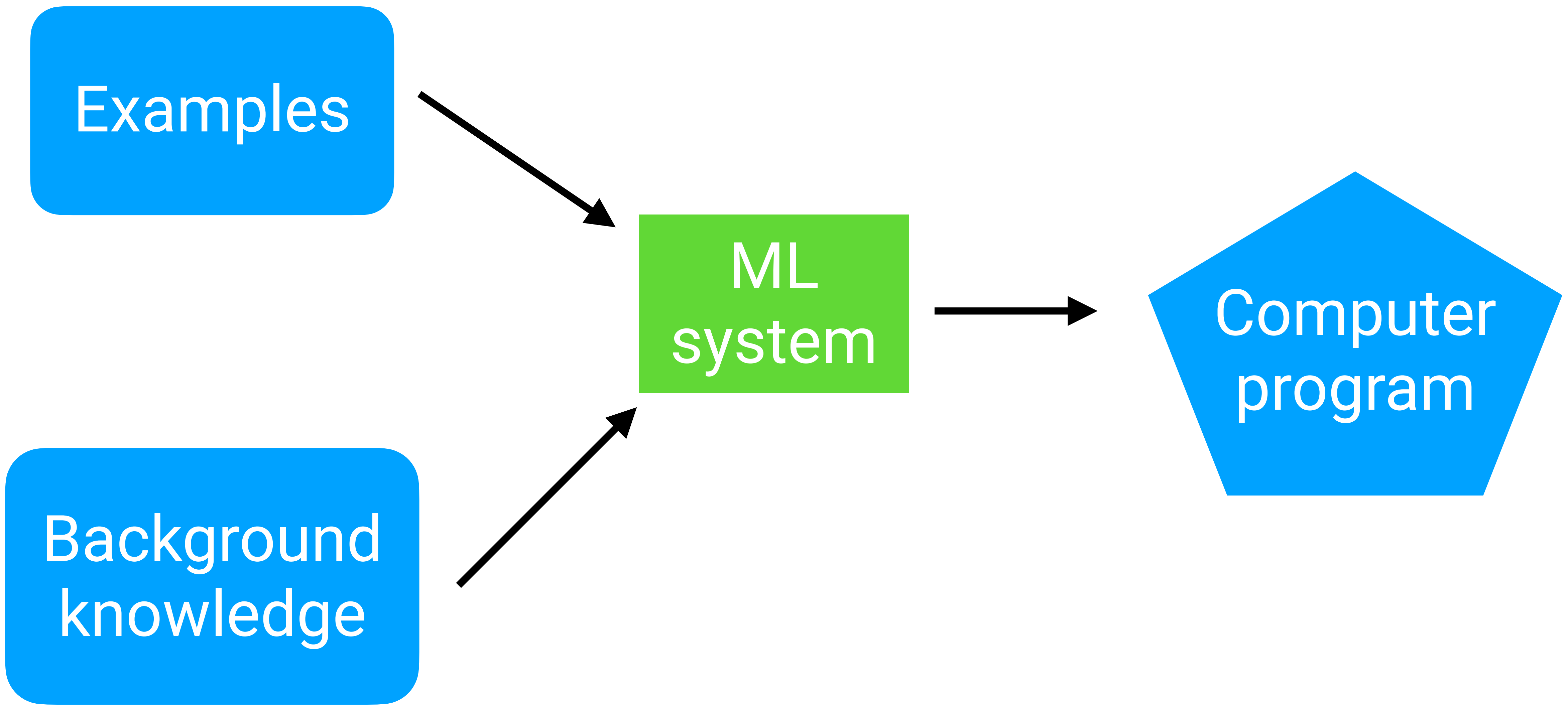
(Popper)

Andrew Cropper and Rolf Morel

What is this talk about?

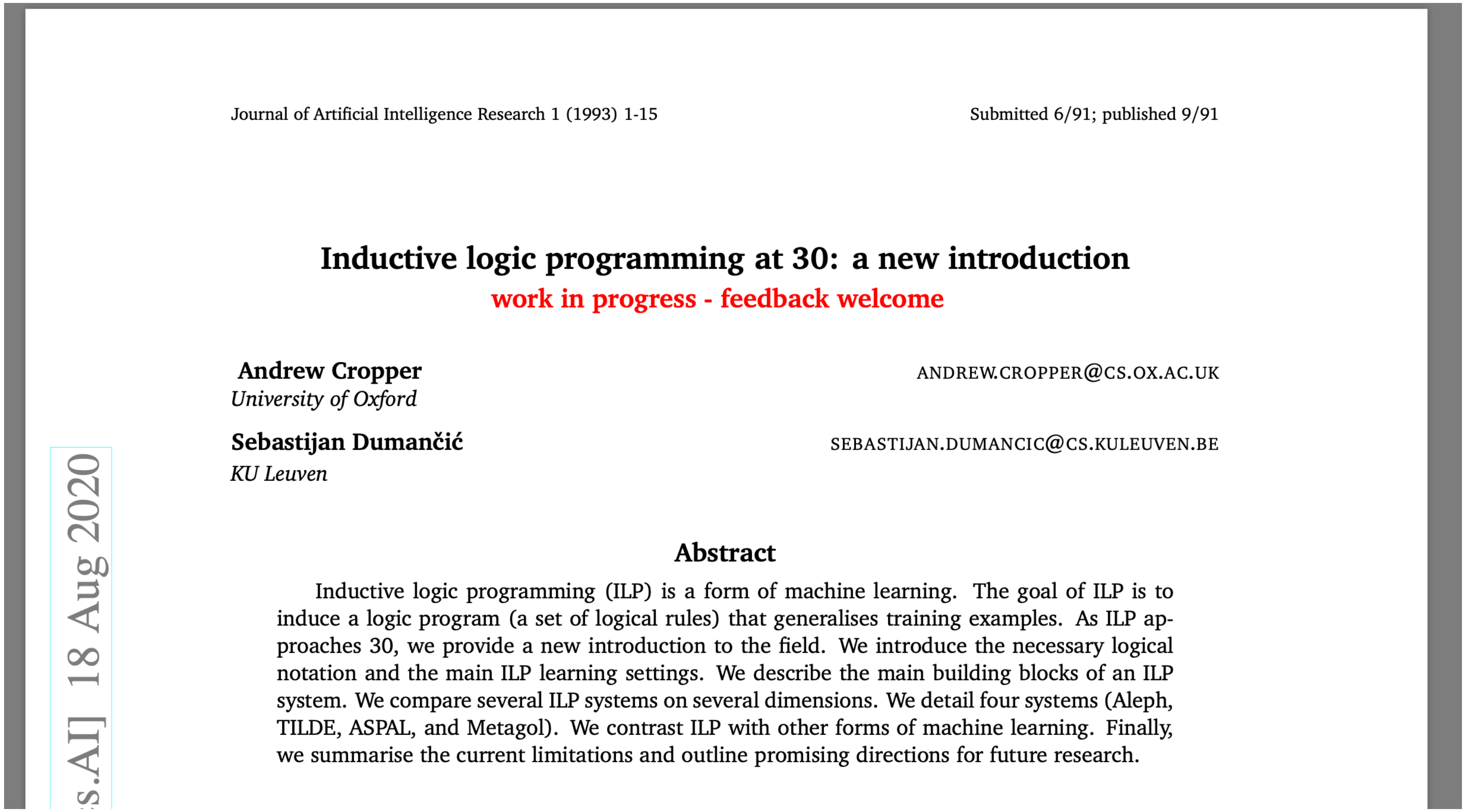
- A new **very simple** form of ILP
- Same functionality as existing approaches
- Much better performance
- Opens up new areas of research

Program induction



Inductive logic programming

A form of ML that uses logic programming to represent data and hypotheses.



Journal of Artificial Intelligence Research 1 (1993) 1-15 Submitted 6/91; published 9/91

Inductive logic programming at 30: a new introduction

work in progress - feedback welcome

Andrew Cropper ANDREW.CROPPER@CS.OX.AC.UK
University of Oxford

Sebastijan Dumančić SEBASTIJAN.DUMANCIC@CS.KULEUVEN.BE
KU Leuven

Abstract

Inductive logic programming (ILP) is a form of machine learning. The goal of ILP is to induce a logic program (a set of logical rules) that generalises training examples. As ILP approaches 30, we provide a new introduction to the field. We introduce the necessary logical notation and the main ILP learning settings. We describe the main building blocks of an ILP system. We compare several ILP systems on several dimensions. We detail four systems (Aleph, TILDE, ASPAL, and Metagol). We contrast ILP with other forms of machine learning. Finally, we summarise the current limitations and outline promising directions for future research.

Examples

input	output
dog	g
sheep	p
chicken	n

representation
last(dog,g)
last(sheep,p)
last(chicken,n)

BK

```
head([H|_],H).
tail([_|T],T).
empty(A).
double(A,B):-A is B+B.
```

Hypothesis

```
last(A,B):-tail(A,C),empty(C),head(A,B).
last(A,B):-tail(A,C),f(C,B).
```

How?

- bottom-up
- top-down
- meta-level

Approaches date back to Banerji (1964), Michaslski (1969), and Plotkin (1971).

Bottom-up
(example driven)

Start with a specific program and generalise it

Advantages	**Disadvantages**
• Fast • Infinite domains	• Optimality • Recursion

LGG (Plotkin,1970), Golem (Muggleton, 1990),
Progol (Muggleton, 1995)

This approach is entirely different to bottom-up approaches described by Solar-Lezama in his lecture notes.

Top-down
(test-driven)

Start with a general program and specialise it

Advantages	**Disadvantages**
• Recursion	• Slow

FOIL (Quinlan 1990), TILDE (Blockeel & De Raedt, 1998), Hyper (Bratko, 1999)

'Reinvented' as test-driven synthesis (Perelman et al, 2014)

Meta-level

Delegate the search to something else

Advantages	Disadvantages
• Recursion • Completeness • Optimality	• Slow • Small domains

ILASP (Law et al, 2014), DILP (Evans and & Grefenstette, 2018), HEXMIL (Kaminski et al., 2019), Apperception (Evans et al., 2019)

Major limitation is that these approaches all precompute all possible rules.

Learning from failures

Advantages	Disadvantages
• Optimality • Completness • Recursion • Infinite domains • Fast • **Simple**	• Noise

This approach is similar but different to CEGIS as we do not produce counter-examples.

Learning from failures

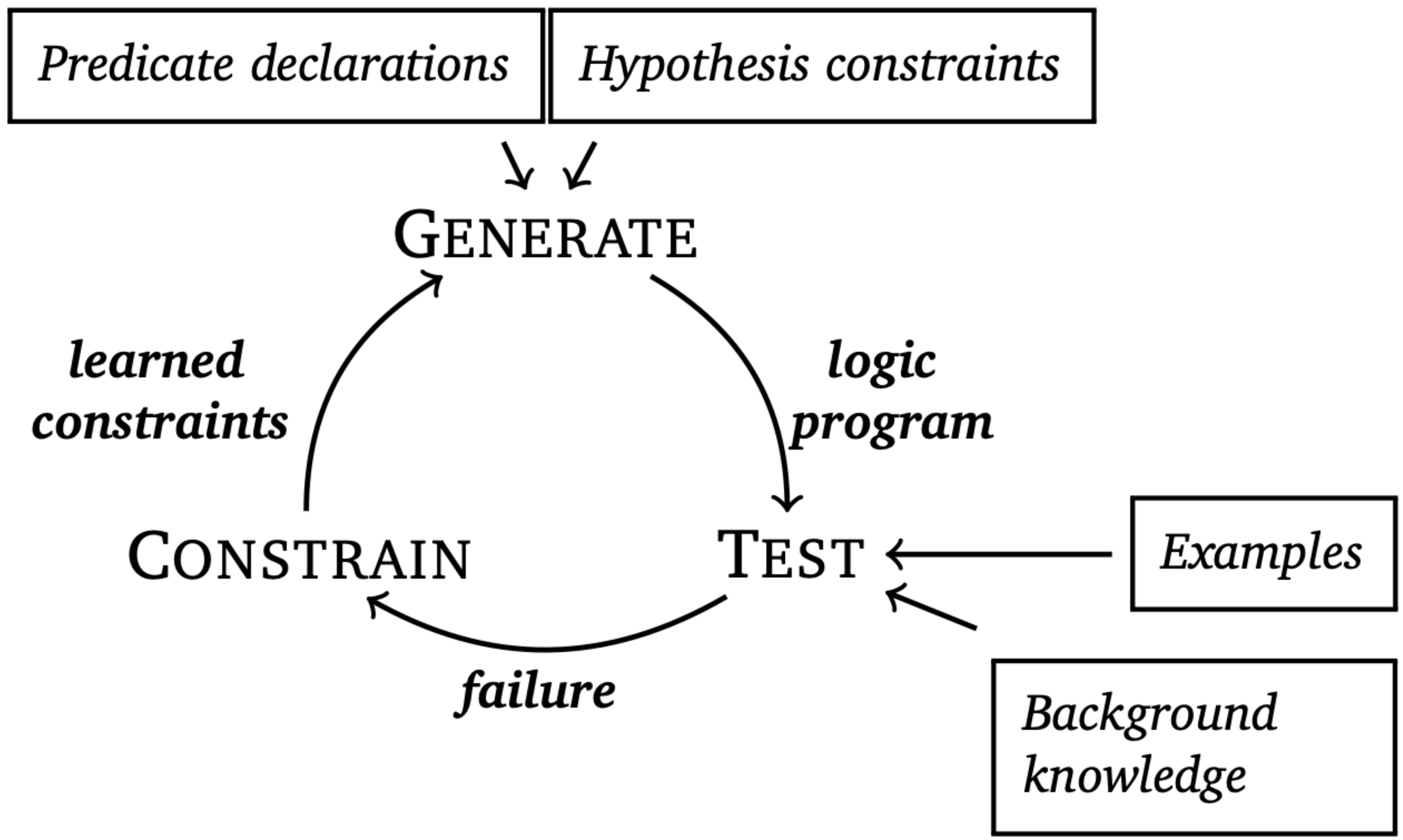
1. Generate
2. Test
3. Constrain

Learning from failures

Automates Karl Popper's idea of falsification:

1. Build a program (form a conjecture)
2. Test it against training examples
3. If it fails (is refuted), **explain why**
4. Use the **explanation** to rule out other programs

MIS (Shaprio, 1983)



(we generate hypothesis constraints, not counter-examples)

input	**output**
laura	a
penelope	e
emma	m
james	e

input	output
laura	a
penelope	e
emma	m
james	e

$$E^+ = \left\{ \begin{array}{l} \texttt{last([l,a,u,r,a],a).} \\ \texttt{last([p,e,n,e,l,o,p,e],e).} \end{array} \right\} \qquad E^- = \left\{ \begin{array}{l} \texttt{last([e,m,m,a],m).} \\ \texttt{last([j,a,m,e,s],e).} \end{array} \right\}$$

$$
\mathcal{H}_1 = \left\{
\begin{array}{l}
\mathsf{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\} \\
\mathsf{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\
\mathsf{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
\mathsf{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
\mathsf{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
\mathsf{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}
\right\}
$$

Hypothesis space is much larger (and can be infinite)

$$h_1 = \{\, \texttt{last(A,B):- head(A,B).} \,\}$$

$$h_1 = \{ \texttt{last(A,B):- head(A,B).} \}$$

input	output	entailed
laura	a	**no**
penelope	e	**no**
emma	m	no
james	e	no

$$h_1 = \{\, \texttt{last(A,B):- head(A,B).} \,\}$$

input	output	entailed
laura	a	**no**
penelope	e	**no**
emma	m	no
james	e	no

H1 is too specific

Prune specialisations

$$\mathscr{H}_1 = \left\{\begin{array}{l}
\mathsf{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\} \\
\mathsf{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\
\mathsf{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
\mathsf{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
\mathsf{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
\mathsf{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}\right\}$$

Prune specialisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l} \cancel{\mathsf{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\ \mathsf{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\ \mathsf{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\ \mathsf{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\ \mathsf{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\ \mathsf{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\ \mathsf{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\ \mathsf{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\} \end{array} \right\}$$

Prune specialisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l} \cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\ \cancel{h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\ h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\ h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\ h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\ h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\ h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\ h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\} \end{array} \right\}$$

Prune specialisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
\cancel{\mathsf{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{\mathsf{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{\mathsf{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\mathsf{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
\mathsf{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
\mathsf{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
\mathsf{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

$$h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\}$$

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

input	output	entailed
laura	a	yes
penelope	e	yes
emma	m	**yes**
james	e	no

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

input	output	entailed
laura	a	yes
penelope	e	yes
emma	m	**yes**
james	e	no

H4 is too general

Prune generalisations

$$
\mathscr{H}_1 = \left\{
\begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}
\right\}
$$

Prune generalisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\cancel{h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\}} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

Prune generalisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\cancel{h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\}} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \cancel{\texttt{last(A,B):- tail(A,C),head(C,B).}} \\ \cancel{\texttt{last(A,B):- reverse(A,C),head(C,B).}} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

Prune generalisations

$$\mathcal{H}_1 = \left\{ \begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\cancel{h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\cancel{h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\cancel{h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\}} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\cancel{h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\}} \\
\cancel{h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\}} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

$$h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\}$$

$$h_5 = \{\ \texttt{last(A,B):- reverse(A,C),head(C,B).}\ \}$$

input	output	entailed
laura	a	yes
penelope	e	yes
emma	m	no
james	e	no

H5 does not fail, so return it

Key ideas

1. Refine the hypothesis space through learned **hypothesis** constraints
2. Decompose the learning problem (i.e. do not just throw the whole problem to a SAT solver)

Hypothesis constraints

- Generalisation
- Specialisation
- Elimination

Constraints are **sound:** they do not prune **optimal** solutions

(see paper for details)

Popper

1. Generate (ASP program)
2. Test (Prolog)
3. Constrain (ASP program)

Generate

Meta-level ASP program, i.e. models are programs

```
% possible clauses
allowed_clause(0..N-1):- max_clauses(N).

% variables
var(0..N-1):- max_vars(N).

% clauses with a head literal
clause(Clause):- head_literal(Clause,_,_,_).

%% head literals
0 {head_literal(Clause,P,A,Vars): head_pred(P,A), vars(A,Vars)} 1:-
    allowed_clause(Clause).

%% body literals
1 {body_literal(Clause,P,A,Vars): body_pred(P,A), vars(A,Vars)} N:-
    clause(Clause), max_body(N).

% variable combinations
vars(1,(Var1,)):- var(Var1).
vars(2,(Var1,Var2)):- var(Var1),var(Var2).
vars(3,(Var1,Var2,Var3)):- var(Var1),var(Var2),var(Var3).
```

Declarative!

Generate

Adding constraints eliminates models and thus programs

```
recursive:- recursive(Clause).

recursive(Clause):- head_literal(Clause,P,A,_), body_literal(Clause,P,A,_).

has_base:- clause(Clause), not recursive(Clause).

% need multiple clauses for recursion
:- recursive(_), not clause(1).

% prevent recursion without a basecase
:- recursive, not has_base.
```

Hard-coded intuitive constraints are important, but they could be learned

Generate

```
head_var(Clause,Var):- head_literal(Clause,_,_,Vars), var_member(Var,Vars).

body_var(Clause,Var):- body_literal(Clause,_,_,Vars), var_member(Var,Vars).

% prevent singleton variables
:- clause_var(Clause,Var), #count{P,Vars: var_in_literal(Clause,P,Vars,Var)} == 1.

% head vars must appear in the body
:- head_var(Clause,Var), not body_var(Clause,Var).

%% type matching
:- var_in_literal(Clause,P,Vars1,Var),var_in_literal(Clause,Q,Vars2,Var),
    var_pos(Var,Vars1,Pos1),var_pos(Var,Vars2,Pos2),
    type(P,Pos1,Type1),type(Q,Pos2,Type2),
    Type1 != Type2.
```

Optional constraints are trivial to express

Test using Prolog

1. Fast
2. Infinite domains
3. Complex data structures

Could use a Datalong engine, or an ASP solver, or something else

Constrain

$$h = \{\ \texttt{last(A,B):- head(A,B).}\ \}$$

```
:-
  head_literal(C0,last,2,(C0V0,C0V1)),
  body_literal(C0,head,2,(C0V0,C0V1)),
  C0V0 != C0V1,clause_size(C0,1).
```

The above is a generalisation constraint

Popper

Algorithm 1 Popper

```
def popper(e+, e-, bk, declarations, constraints, max_literals):
  num_literals = 1
  while num_literals ≤ max_literals:
    program = generate(declarations, constraints, num_literals)
    if program == 'space_exhausted':
      num_literals += 1
      continue
    outcome = test(e+, e-, bk, program)
    if outcome == ('all_positive', 'none_negative')
      return program
    constraints += learn_constraints(program, outcome)
  return {}
```

Uses clingo's multi-shot solving to remember state

Popper

	Progol	**Metagol**	**ILASP**	**∂ILP**	**Popper**
Hypotheses	Normal	Definite	ASP	Datalog	Definite
Language bias	Modes	Metarules	Modes	Templates	Declarations
Predicate invention	No	**Yes**	Partly	Partly	No
Noise handling	**Yes**	No	**Yes**	**Yes**	No
Recursion	Partly	**Yes**	**Yes**	**Yes**	**Yes**
Optimality	No	**Yes**	**Yes**	**Yes**	**Yes**
Infinite domains	**Yes**	**Yes**	No	No	**Yes**
Hypothesis constraints	No	No	No	No	**Yes**

No sketches / templates, such as in Metagol, DILP, Sketch, or SyGuS

Does it work?

Q1. Can constraints improve learning performance, i.e. does it outperform pure enumeration?

Q2. Can Popper outperform SOTA ILP systems?

Q3. How well does Popper scale?

Run on a MacBook pro on a single CPU with a timeout of two minutes

Primorials

Purposely simple experiment to test the claims

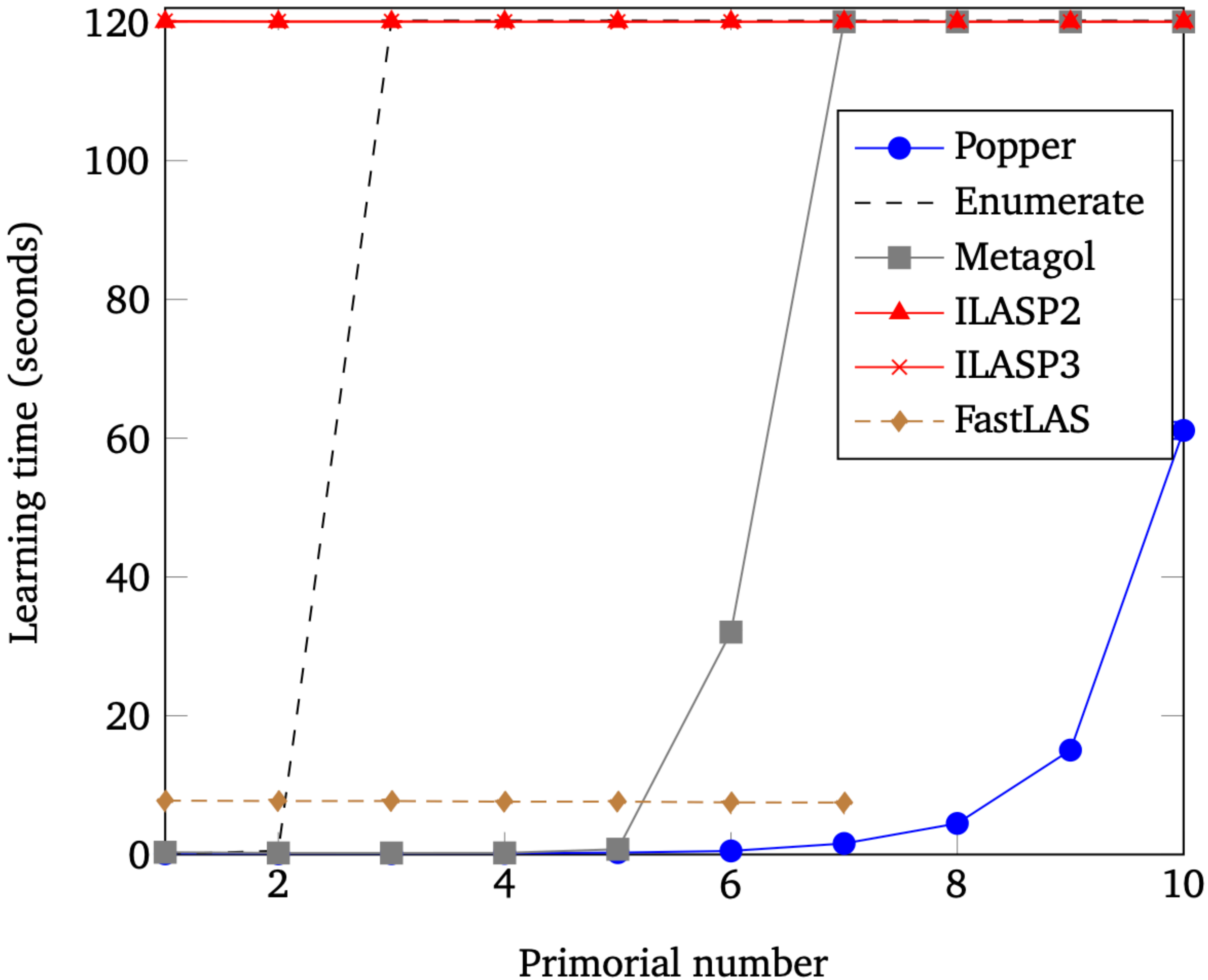
$$p_n\# \equiv \prod_{k=1}^{n} p_k$$

$$p_5\# = 2 \times 3 \times 5 \times 7 \times 11 = 2310$$

```
primorial5(A):- div2(A),div3(A),div5(A),div7(A),div11(A).
```

Hypothesis space contains about 10^{13} programs

Primorials

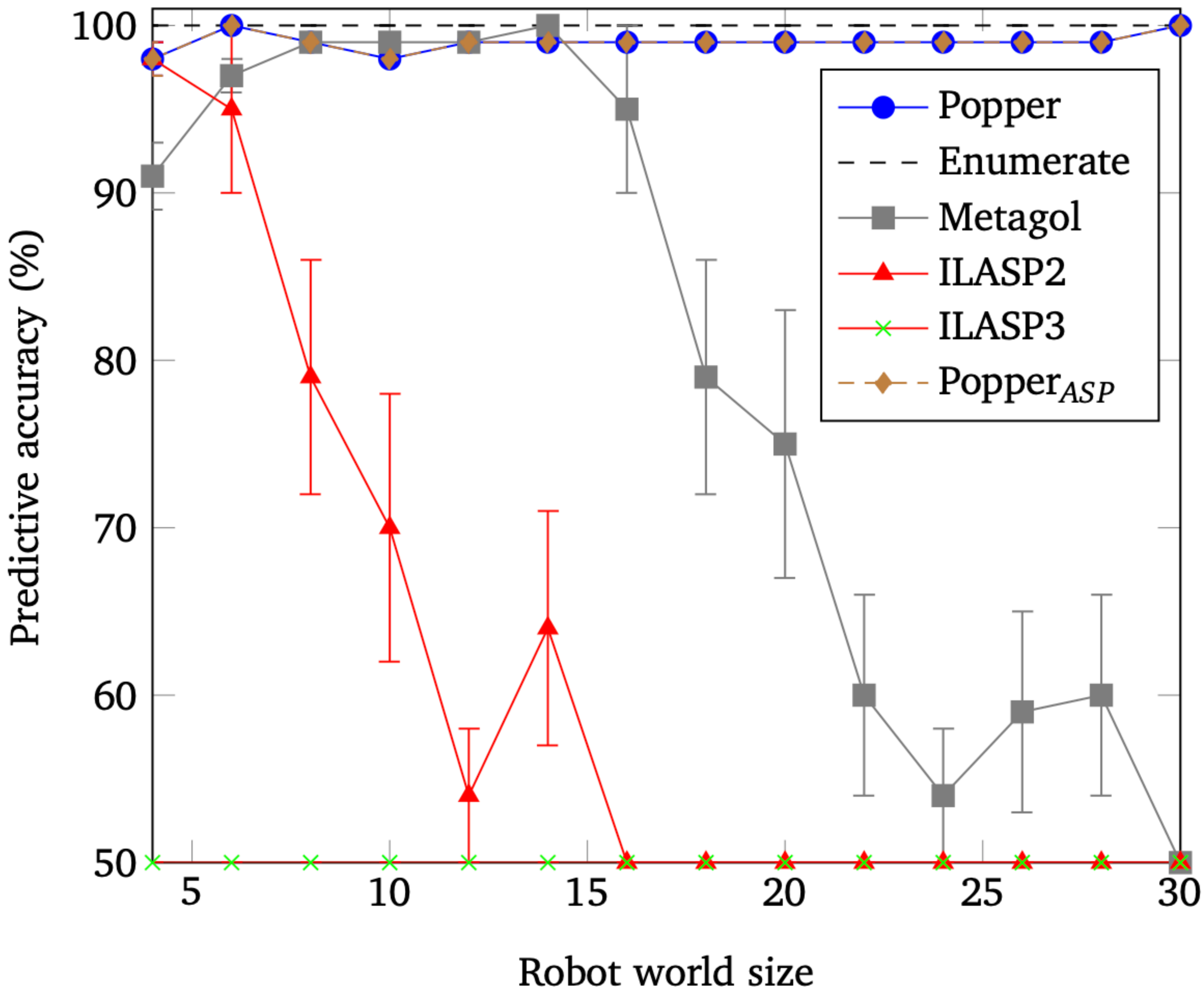


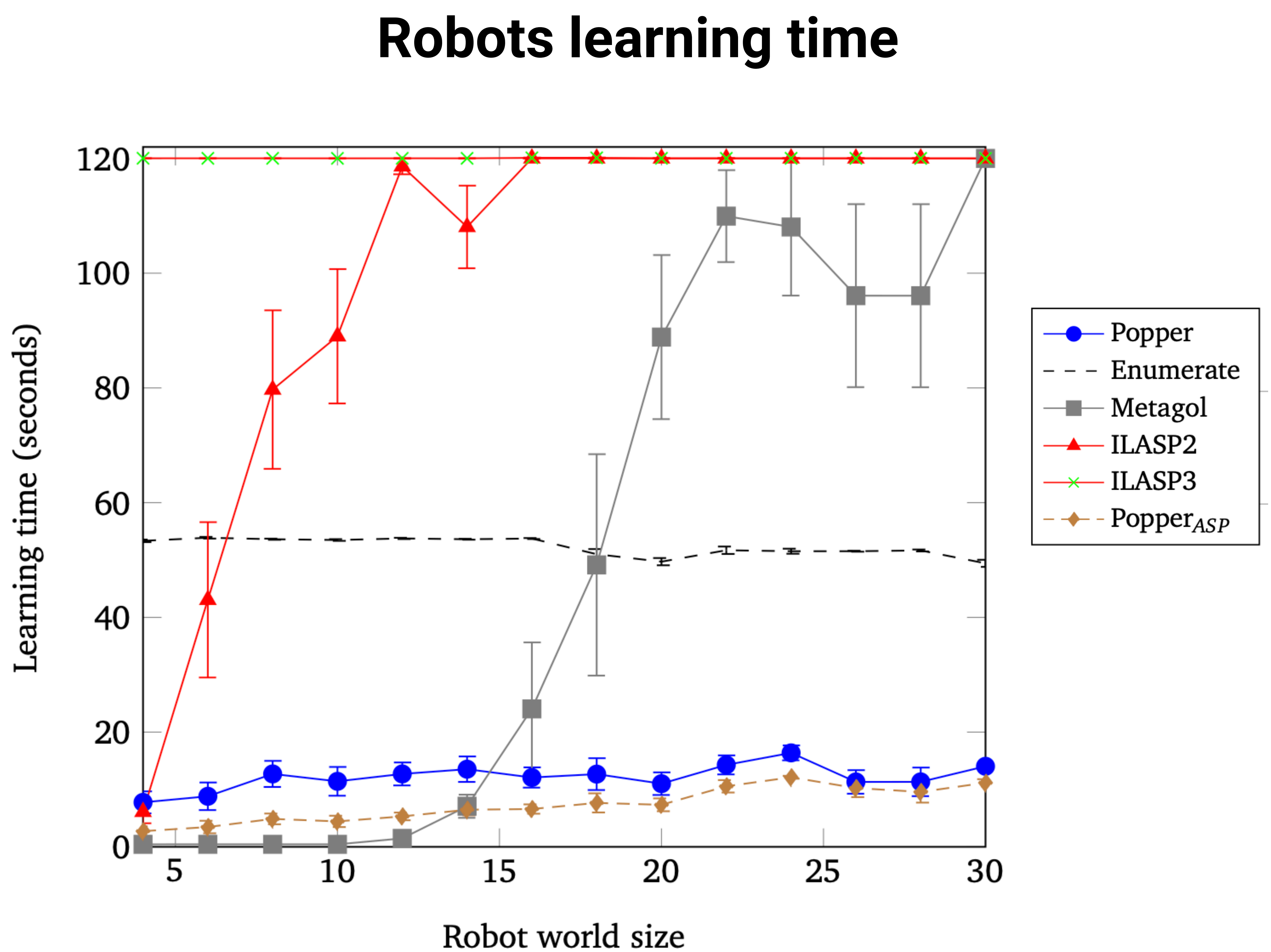
(program size)

Robots

- $n \times n$ grid world.
- keep moving upwards until you cannot move upwards any more
- 5 positive and 5 negative examples

Robots accuracy




Robots learning time
Learning time (seconds)
Robot world size
0
20
40
60
80
100
120
5
10
15
20
25
30
Popper
Enumerate
Metagol
ILASP2
ILASP3
Popper$_{ASP}$

Programming puzzles

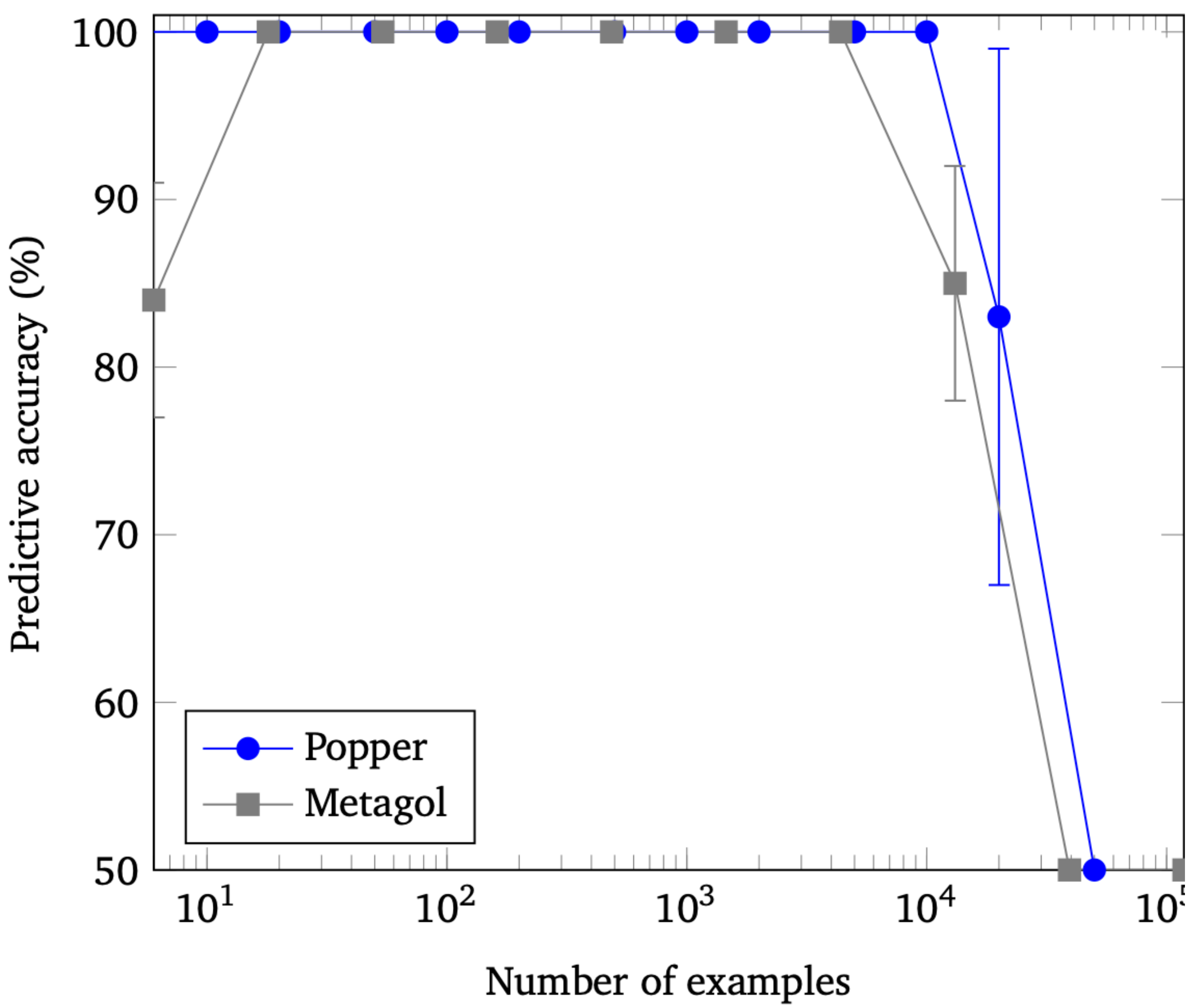
Name	Description	Example solution
`addhead`	Prepend the head three times	addhead(A,B):—head(A,C),cons(C,A,D),cons(C,D,E),cons(C,E,B).
`dropk`	Drop the first k elements	dropk(A,B,C):—one(B),tail(A,C). dropk(A,B,C):—tail(A,D),decrement(B,E),dropk(D,E,B).
`droplast`	Drop the last element	droplast(A,B):—tail(A,B), tail (B,C),empty(C). droplast(A,B):—tail(A,C),droplast(C,D),head(A,E),cons(E,D,B).
`evens`	Check all elements are even	evens(A):—empty(A). evens(A):—even(A),tail(A,C),evens(C).
`finddup`	Find duplicate elements	finddup(A,B):—head(A,B),tail(A,C),member(B,C). finddup(A,B):—tail(A,C),finddup(C,B).
`last`	Last element	last (A,B):—tail(A,C),empty(C),head(A,B). last (A,B):—tail(A,C), last (C,B).
`len`	Calculate list length	len(A,B):—empty(A),zero(B). len(A,B):—tail(A,C),len(C,D),succ(D,B).
`member`	Member of a list	member(A,B):—head(A,B). member(A,B):—tail(A,C),member(C,B).
`sorted`	Check list is sorted	sorted(A):—empty(A). sorted(A):—head(A,B),tail(A,C),head(C,D),geq(D,B),sorted(C).
`threesame`	First three elements are identical	threesame(A):—head(A,B),tail(A,C),head(C,B),tail(C,D),head(D,B).

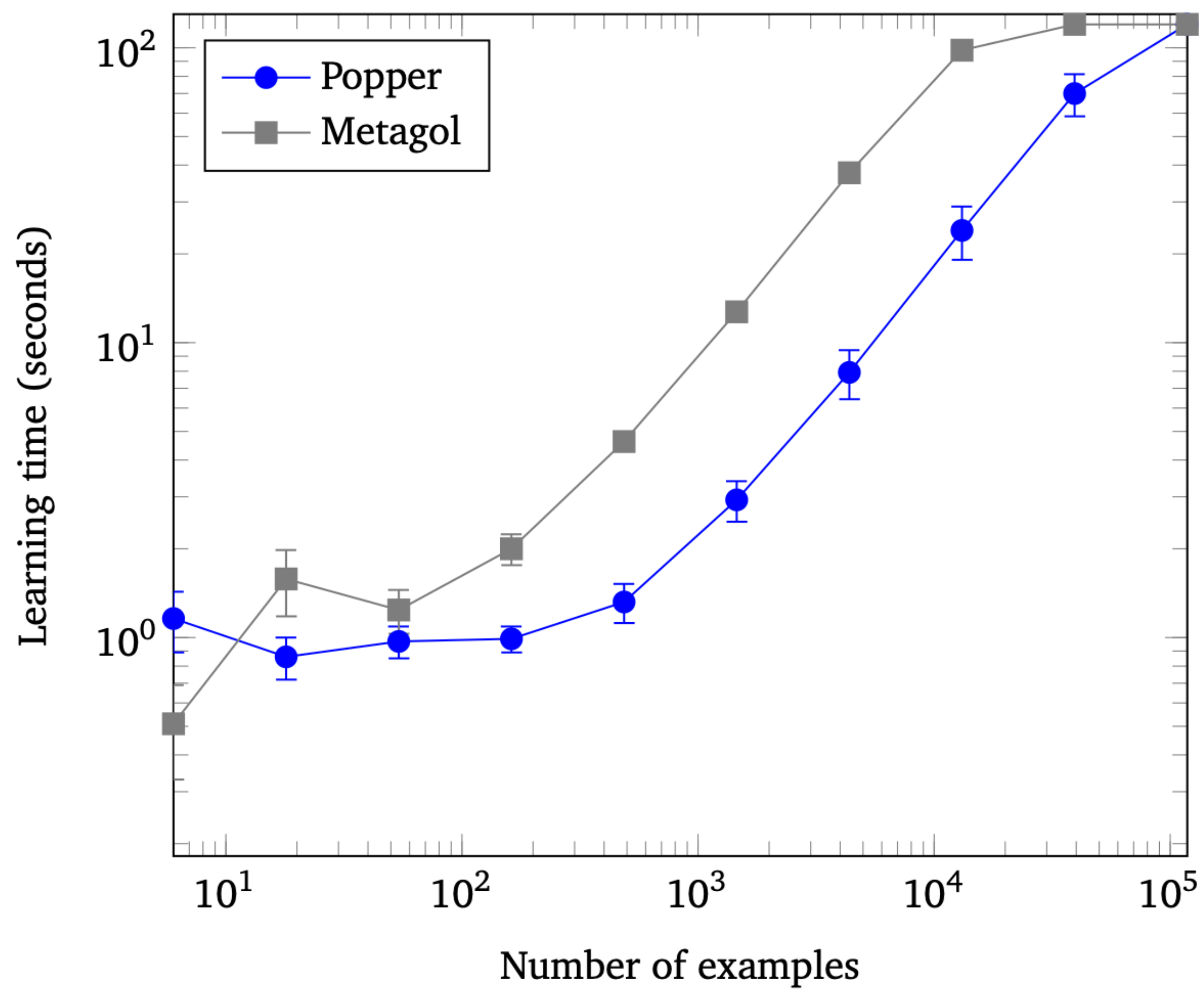
Programming puzzles

	Accuracies			Times		
Name	**Popper**	**Enumerate**	**Metagol**	**Popper**	**Enumerate**	**Metagol**
`addhead`	**100** ± 0	**100** ± 0	50 ± 0	**1** ± 0	3 ± 0	120 ± 0
`dropk`	**100** ± 0	50 ± 0	50 ± 0	**1** ± 0	120 ± 0	120 ± 0
`droplast`	**100** ± 0	50 ± 0	50 ± 0	**39** ± 4	120 ± 0	120 ± 0
`evens`	**100** ± 0	50 ± 0	55 ± 5	**4** ± 0.41	120 ± 0	109 ± 11
`finddup`	99 ± 0	80 ± 0	**100** ± 0	13 ± 2	57 ± 18	**2** ± 0
`last`	**100** ± 0	**100** ± 0	**100** ± 0	0.72 ± 0.11	**0.55** ± 0.08	0.83 ± 0.09
`len`	**100** ± 0	50 ± 0	50 ± 0	**7** ± 1	120 ± 0	120 ± 0
`member`	**100** ± 0	**100** ± 0	75 ± 8	**0.14** ± 0.01	2 ± 0.01	0.42 ± 0.01
`sorted`	**100** ± 6	50 ± 0	50 ± 0	**77** ± 7	120 ± 0	120 ± 0
`threesame`	**99** ± 0	**99** ± 0	**99** ± 0	**0.32** ± 0.02	0.47 ± 0.04	0.35 ± 0.06

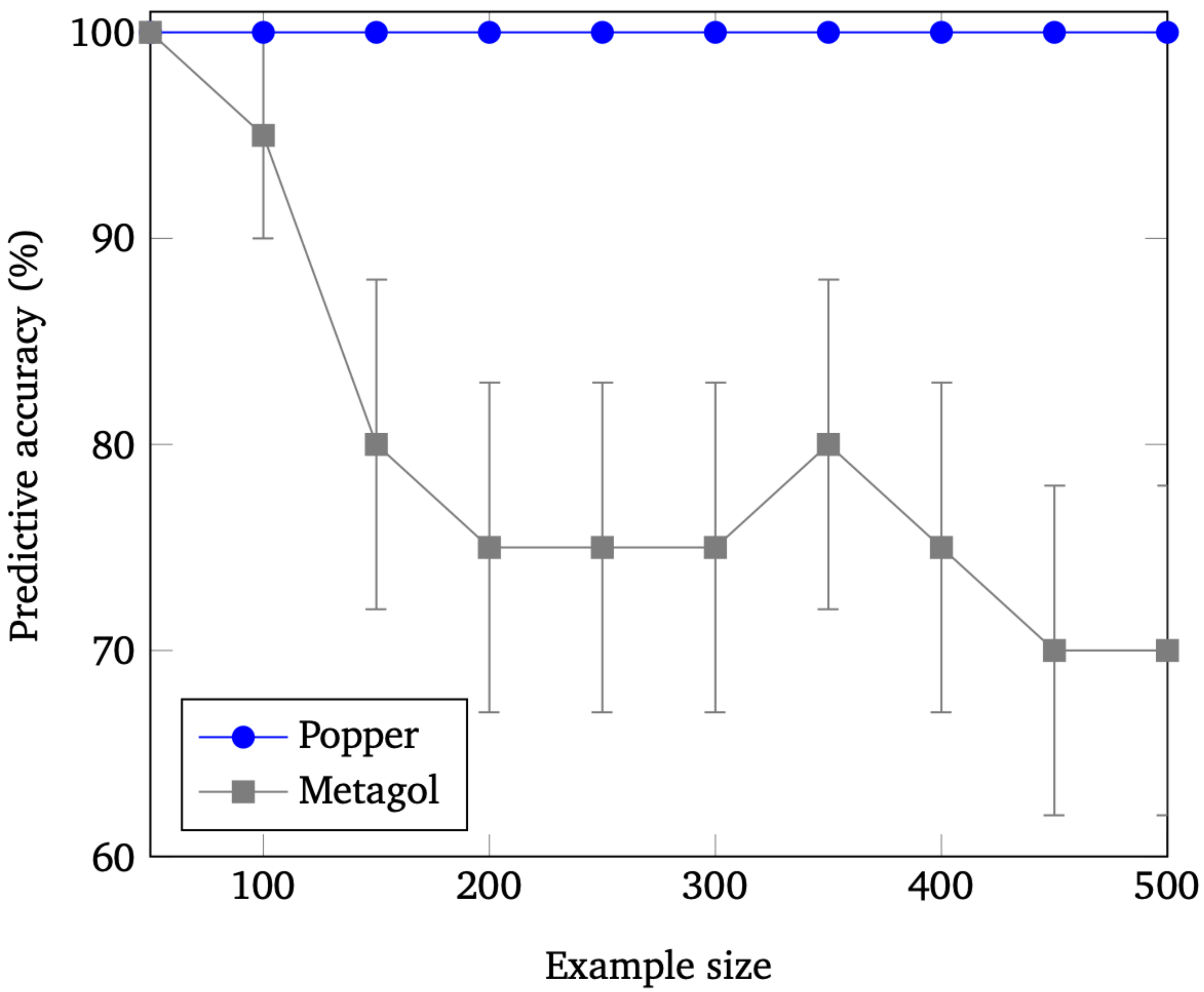
(we have since cut Popper learning times by 1/2)

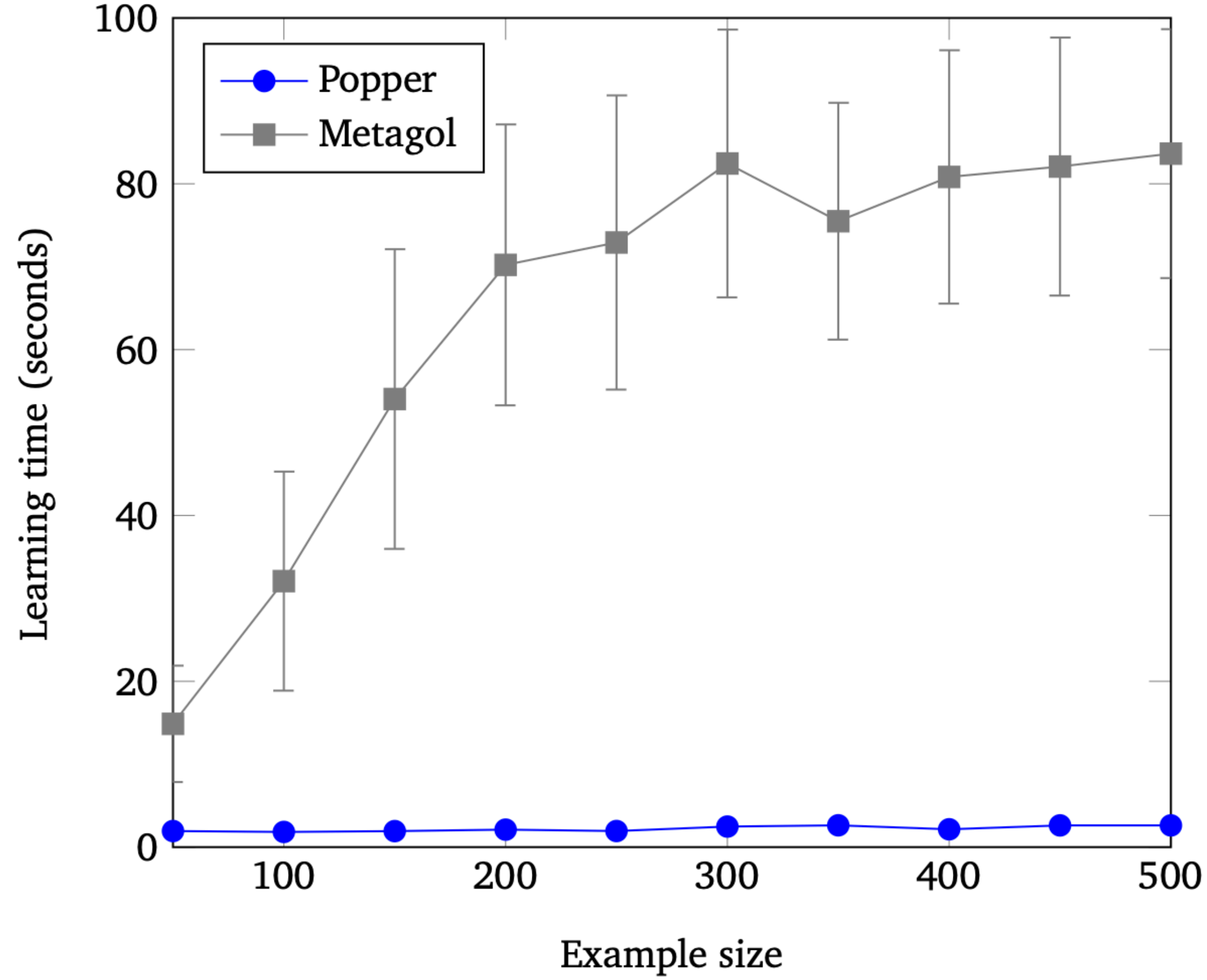
Scalability: number of examples





Scalability: size of examples





Sensitivity

- the maximum number of unique variables in a clause
- the maximum number of body literals allowed in a clause
- the maximum number of clauses allowed in a hypothesis

Bottleneck is the number of variables in a clause

Conclusions

Simplicity: LFF is a simple form of ILP that does need metarules, strong priors, etc.

Performance: Popper significantly outperforms SOTA approaches.

Feature rich: Popper supports recursion, infinite domains, and learning optimal programs.

Limitations

- Noise
- Negation
- **Predicate invention**

Future work

Constraints: What can we learn from failures?

Search: Proving unsatisfiability at each program size is the major bottleneck. Would it be better to try larger programs to learn more from failing?

Parallelisation: How to search in parallel?

Applications: IGGP and ARC datasets

Papers

Learning programs by learning from failures. Cropper and Morel. Under review.

Turning 30: new ideas in inductive logic programming. Cropper, Dumančić, and Muggleton. IJCAI2020.

Inductive logic programming at 30: a new introduction. Cropper and Dumančić. In preparation.