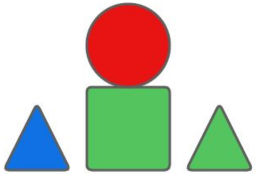
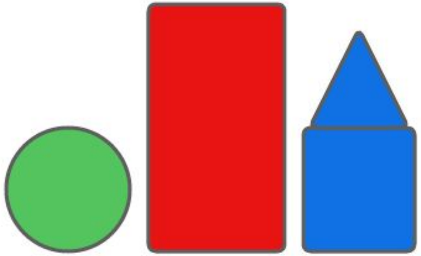
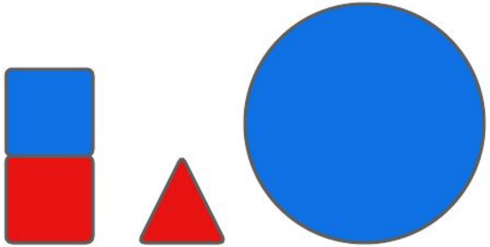
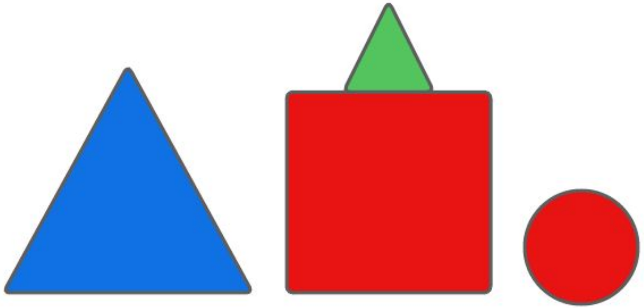
Learning logic programs with constraint programming

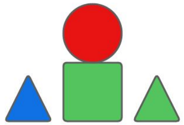
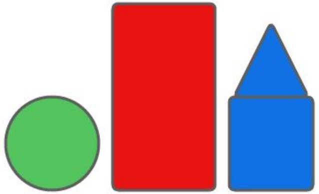
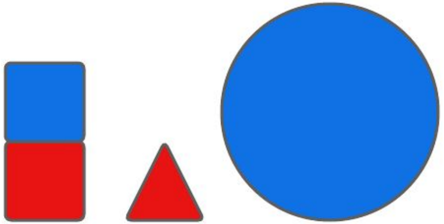
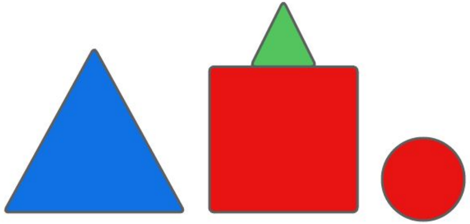
Céline Hocquette
University of Oxford / Southampton

UK Research and Innovation

University of Oxford

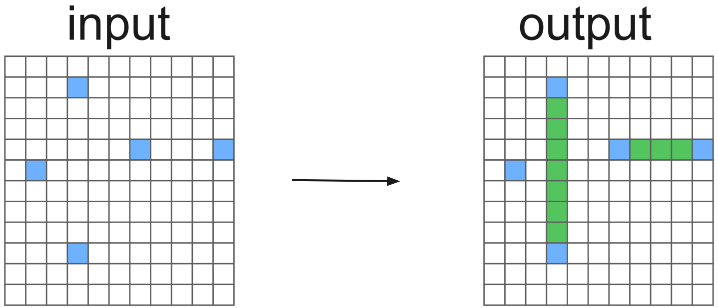
University of Southampton

Positive structures	Negative structures
	
	

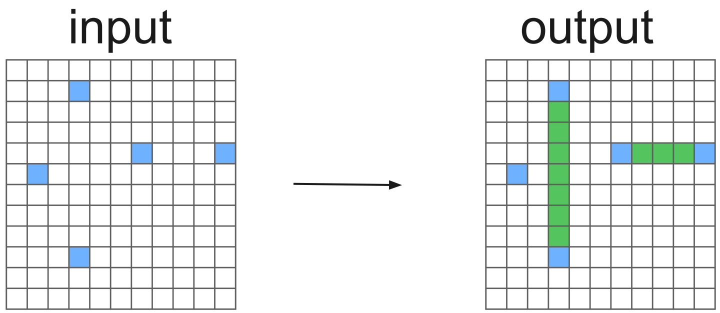
Positive structures	Negative structures
	
	

There must be a red piece in contact with a square piece

Abstraction and Reasoning Corpus (ARC) [Chollet, 2019]

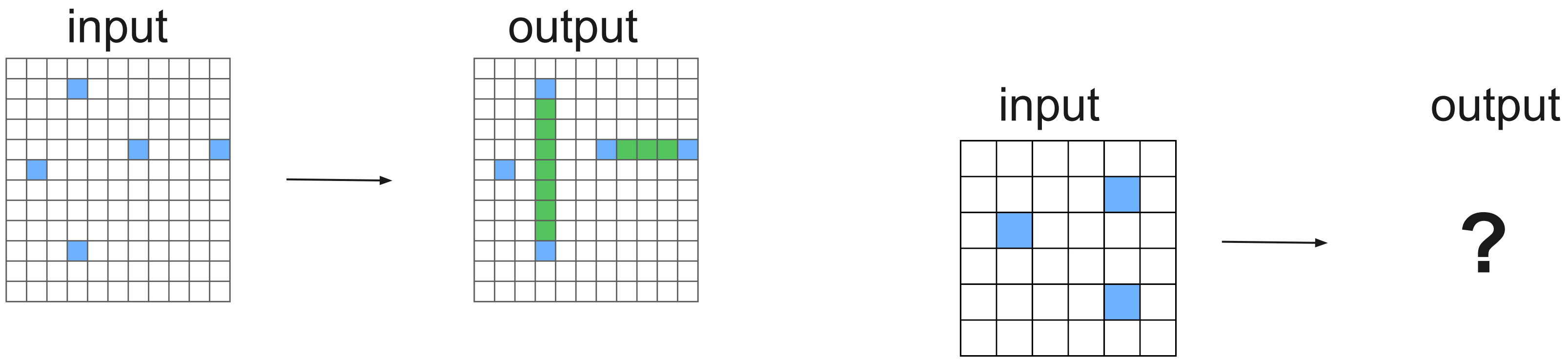


Abstraction and Reasoning Corpus (ARC) [Chollet, 2019]



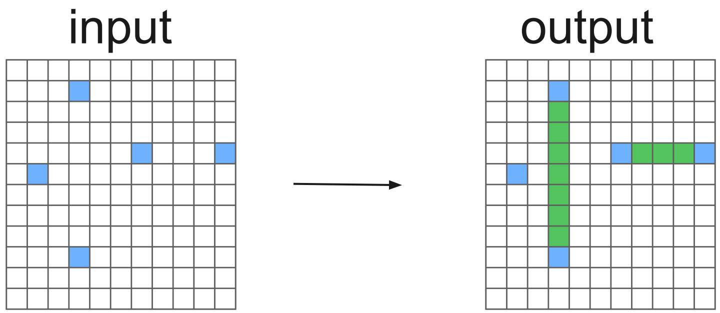
Color in green pixels in between two blue pixels

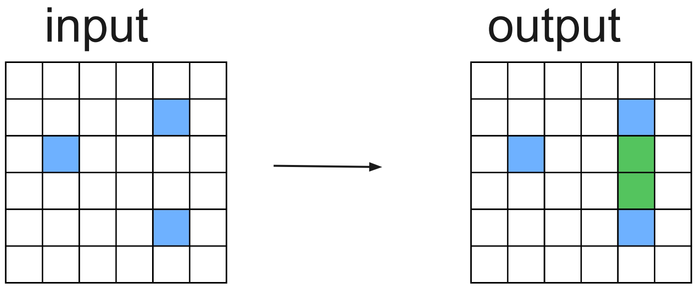
Abstraction and Reasoning Corpus (ARC) [Chollet, 2019]



Color in green pixels in between two blue pixels

Abstraction and Reasoning Corpus (ARC) [Chollet, 2019]





Inductive Logic Programming

Inductive Logic Programming

a form of program synthesis based on logic

Inductive Logic Programming

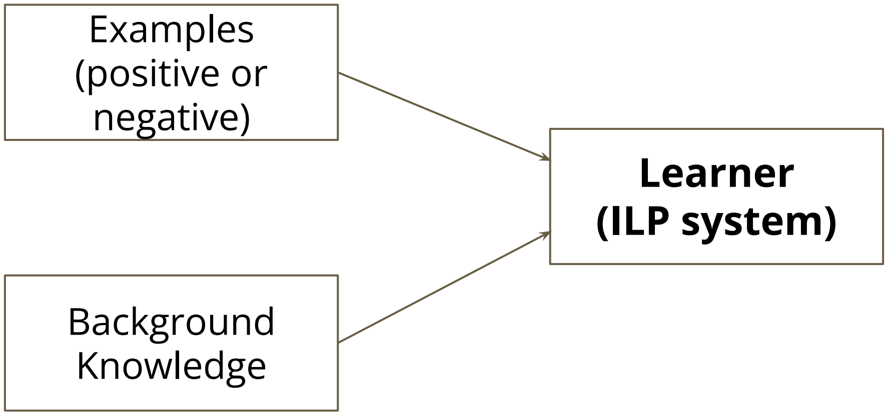
Examples
(positive or
negative)

Inductive Logic Programming

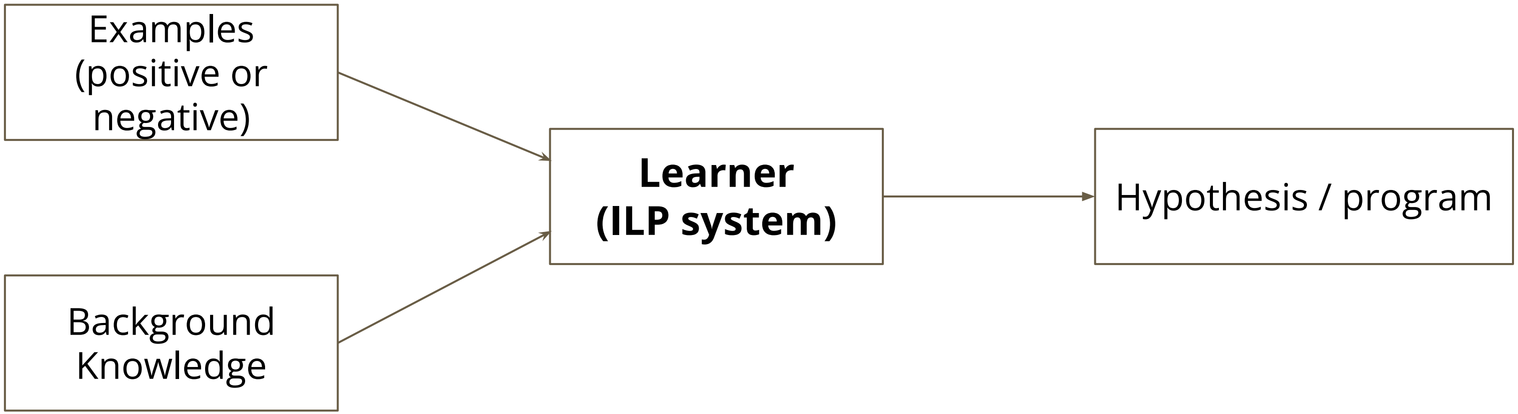
Examples (positive or negative)

Background Knowledge

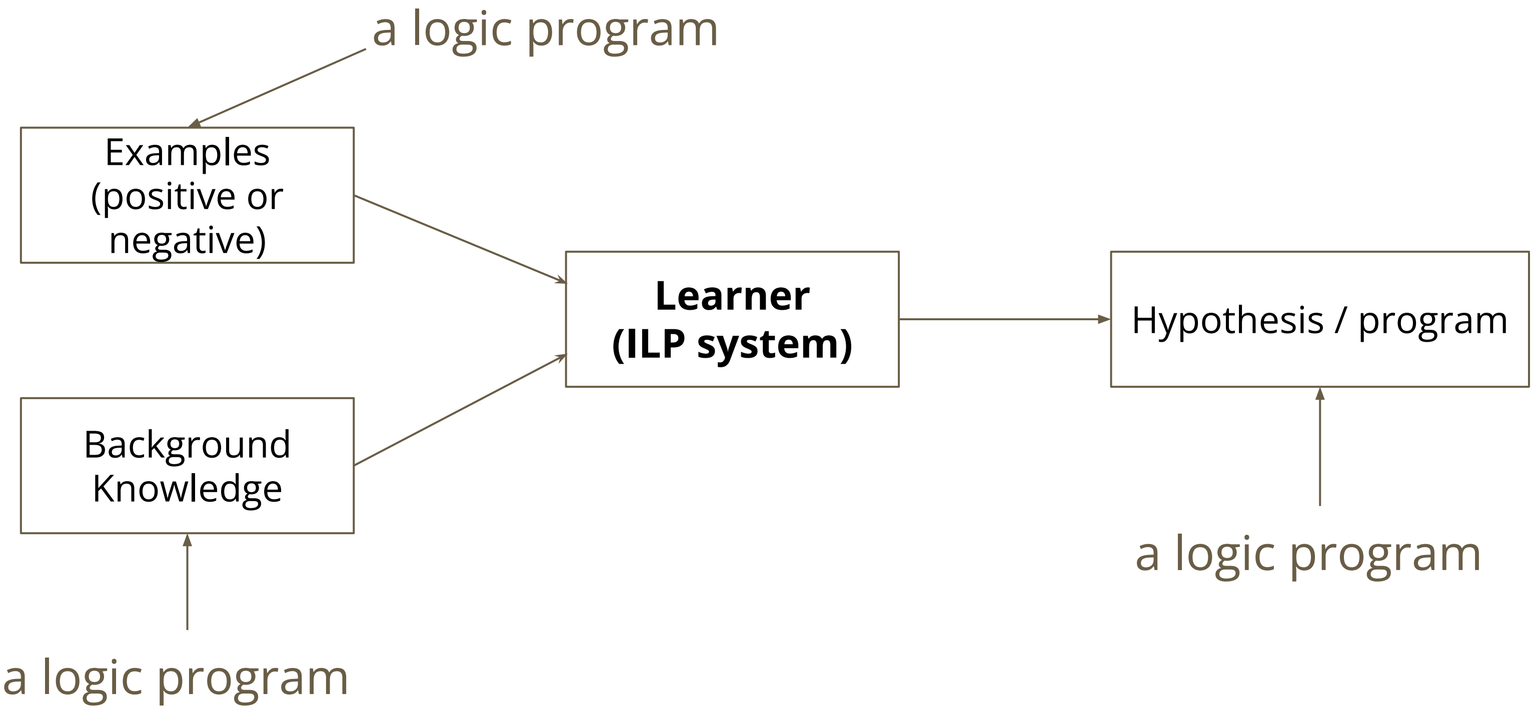
Inductive Logic Programming



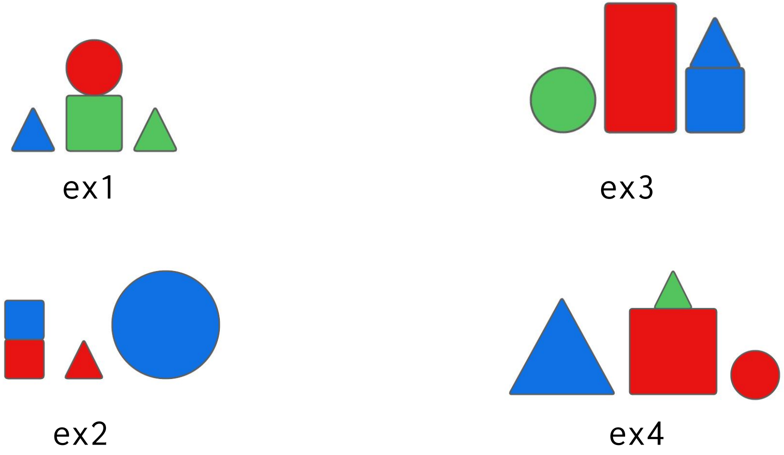
Inductive Logic Programming



Inductive Logic Programming

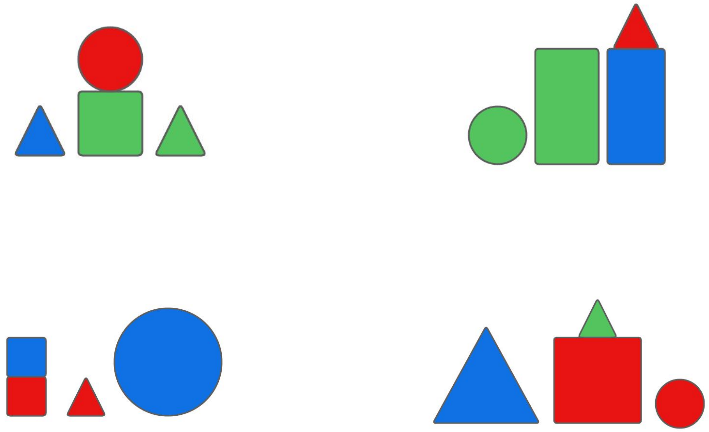


Positive examples	Negative examples
zendo(ex1). zendo(ex2).	zendo(ex3). zendo(ex4).



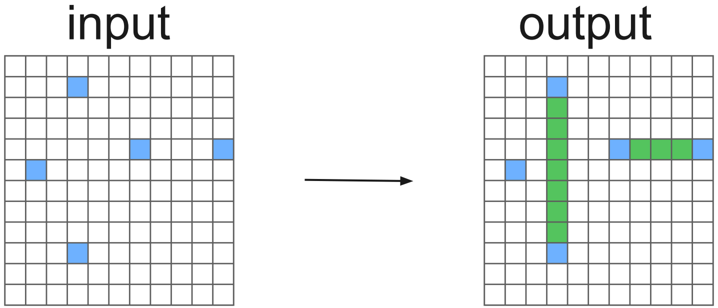
Background Knowledge

```
piece(ex1, p1).
piece(ex1, p2).
piece(ex1, p3).
piece(ex1, p4).
blue(p1).
triangle(p1).
size(p1, 2).
small(2).
red(p2).
round(p2).
triangle(p4).
contact(p2, p3).
on(p2, p3).
right(p4, p3).
left(p1, p2).
…
```



Hypothesis

```
zendo(Structure) ←
    piece(Structure,Piece1),
    red(Piece1),
    contact(Piece1,Piece2),
    square(Piece2).
```



Hypothesis
out(X,Y,blue) ← in(X,Y,blue).
out(X,Y,green) ← in(X1,Y,blue), in(X2,Y,blue), X1<X<X2.
out(X,Y,green) ← in(X,Y1,blue), in(X,Y2,blue), Y1<Y<Y2.

Why ILP?

Why ILP?

- high generalisation ability

Why ILP?

- high generalisation ability
- learn from small amount of data

Why ILP?

- high generalisation ability
- learn from small amount of data
- learn from highly relational data

Why ILP?

- high generalisation ability
- learn from small amount of data
- learn from highly relational data
- learn explainable and verifiable models

Challenge

Challenge

hypothesis space = the set of all programs which may be learned by the learner

Large hypothesis spaces!

Challenge

hypothesis space = the set of all programs which may be learned by the learner

Large hypothesis spaces!

Zendo: 10^8 hypotheses with 1 rule and at most 6 variables and at most 6 literals

In this presentation

1. An approach that formulates the ILP problem as a CP problem
2. Discovering constraints
3. Learning programs with many rules
4. Learning programs with big rules

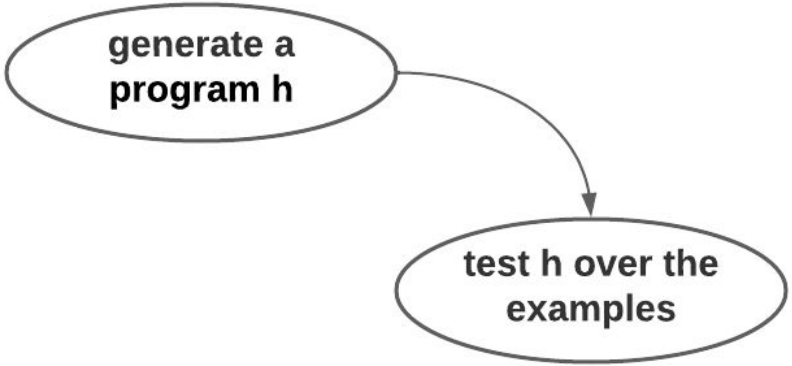
1 - ILP as CP

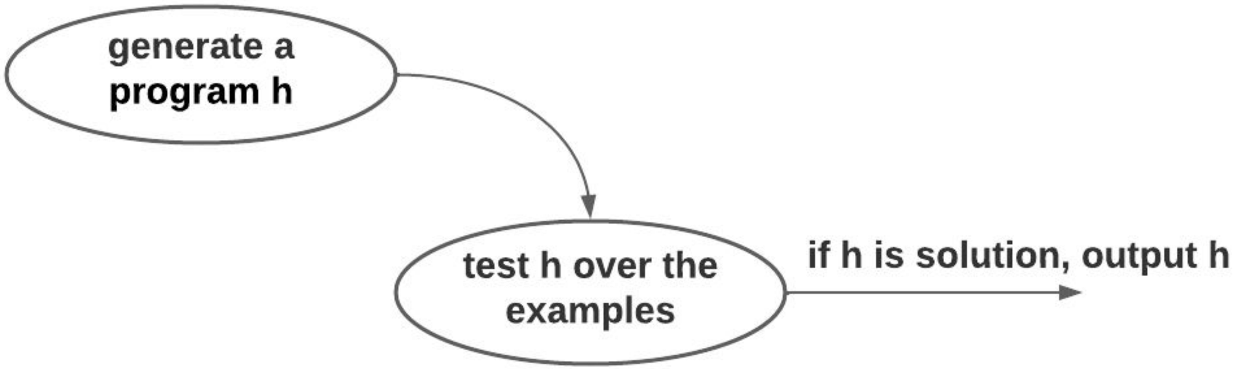
Popper: an ILP system based on CP

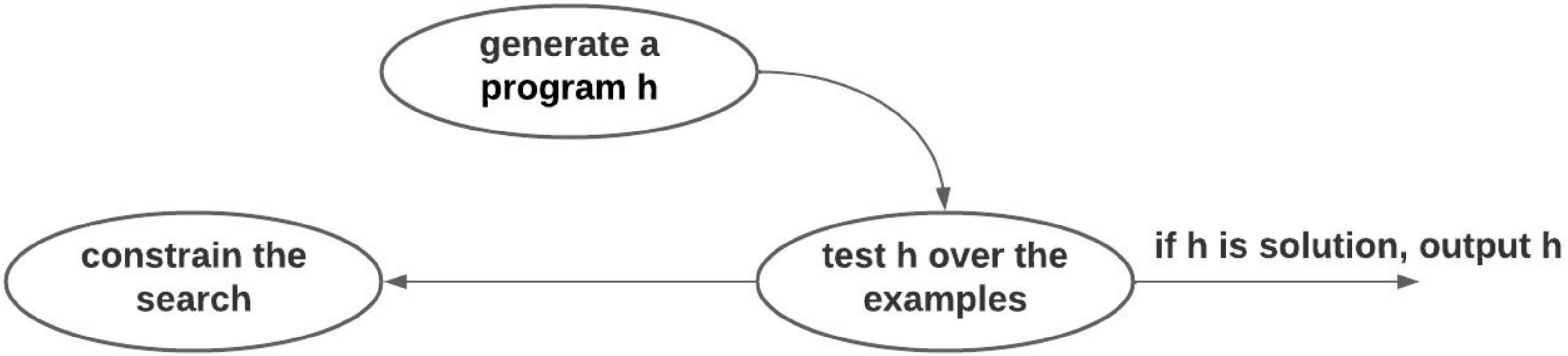
https://github.com/logic-and-learning-lab/Popper

generate a
program h

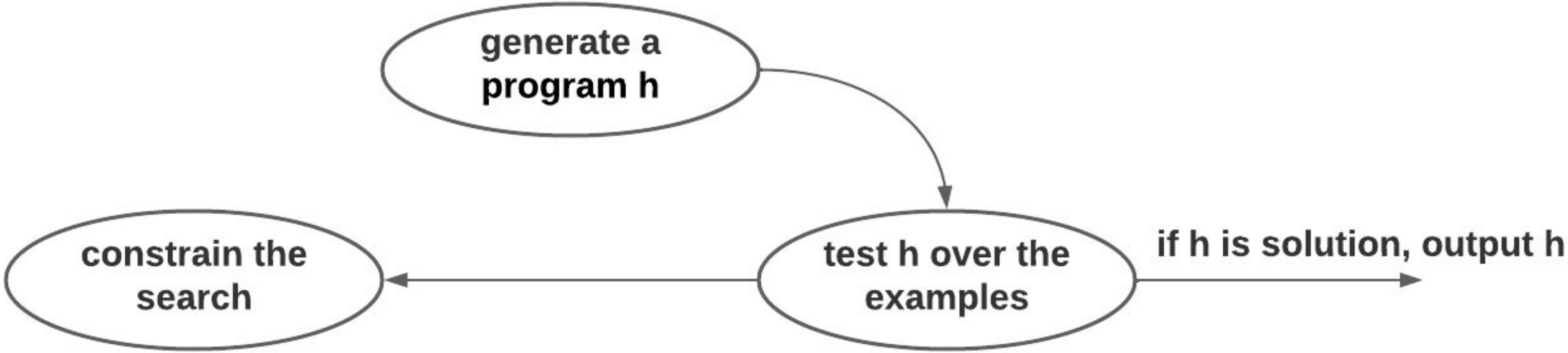
```
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1).
```


generate a program h
test h over the examples


generate a program h
test h over the examples
if h is solution, output h

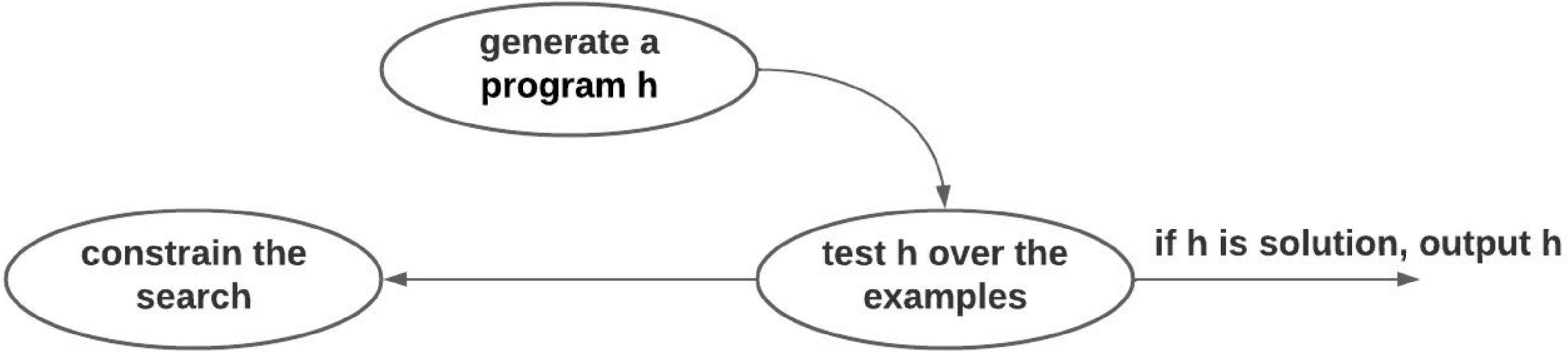

generate a program h
test h over the examples
if h is solution, output h
constrain the search

```
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1).
```



entails 0 positive examples

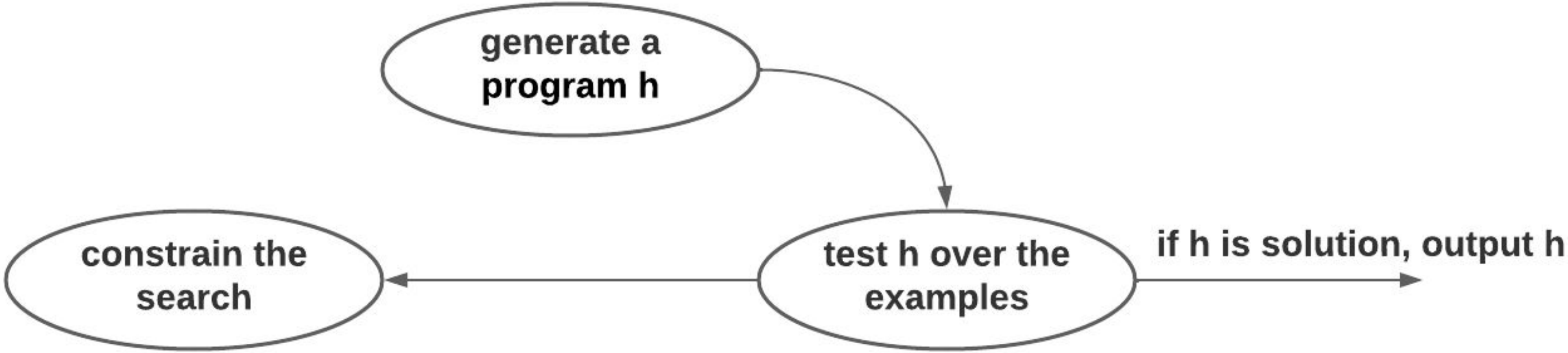
```
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1).
```



we prune specialisations of h

entails 0 positive examples

```
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1).
```

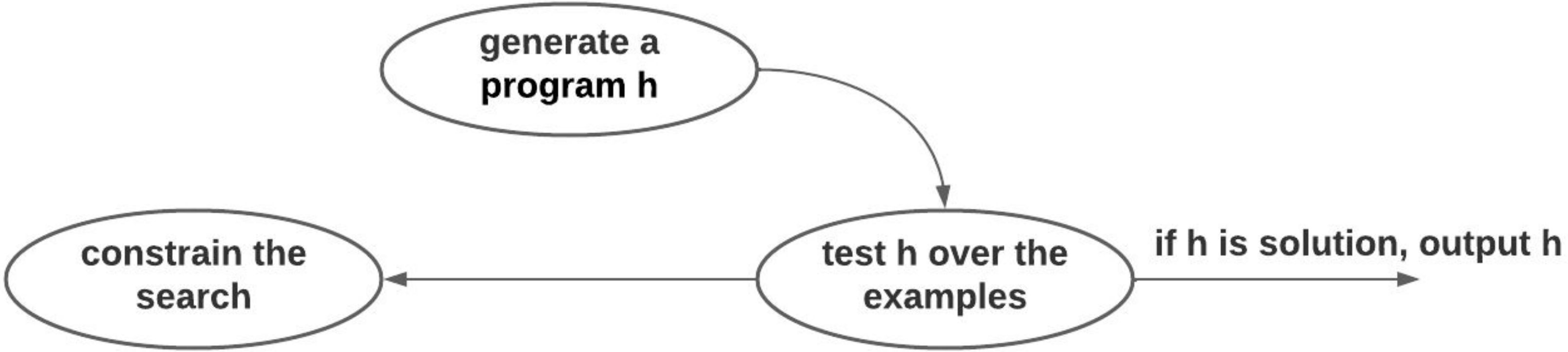


we prune specialisations of h

entails 0 positive examples

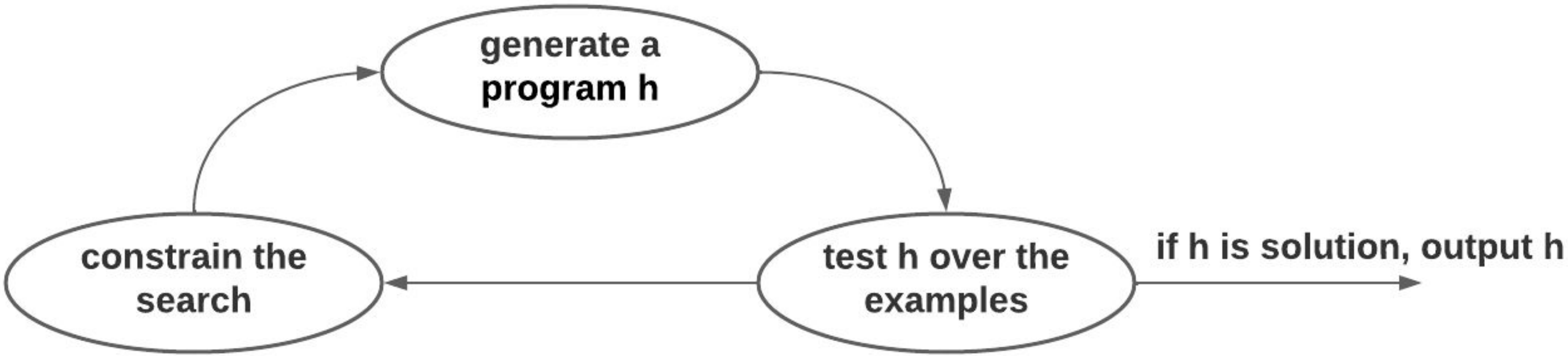
```
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1),small(Piece1).
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1),round(Piece1).
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1),contact(Piece1,Piece2),red(Piece2).
```

```
zendo(Structure) ← piece(Structure,Piece1),yellow(Piece1).
```

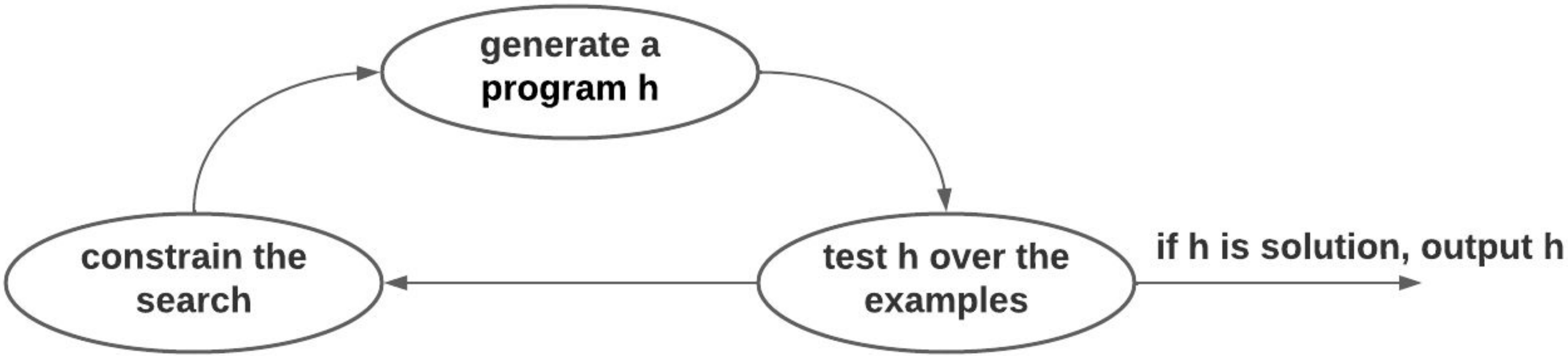


we prune generalisations of h

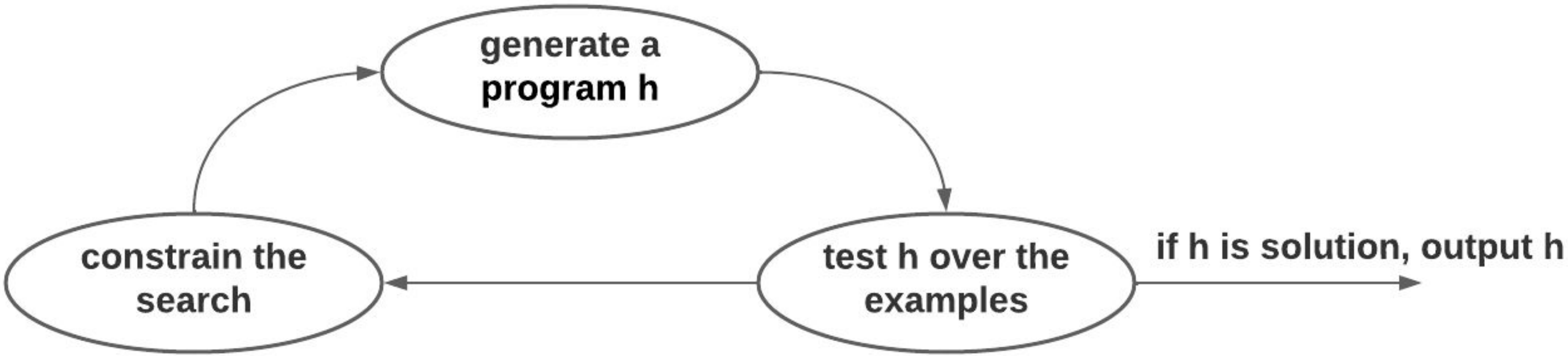
entails 1 negative example


generate a
program h
test h over the
examples
if h is solution, output h
constrain the
search

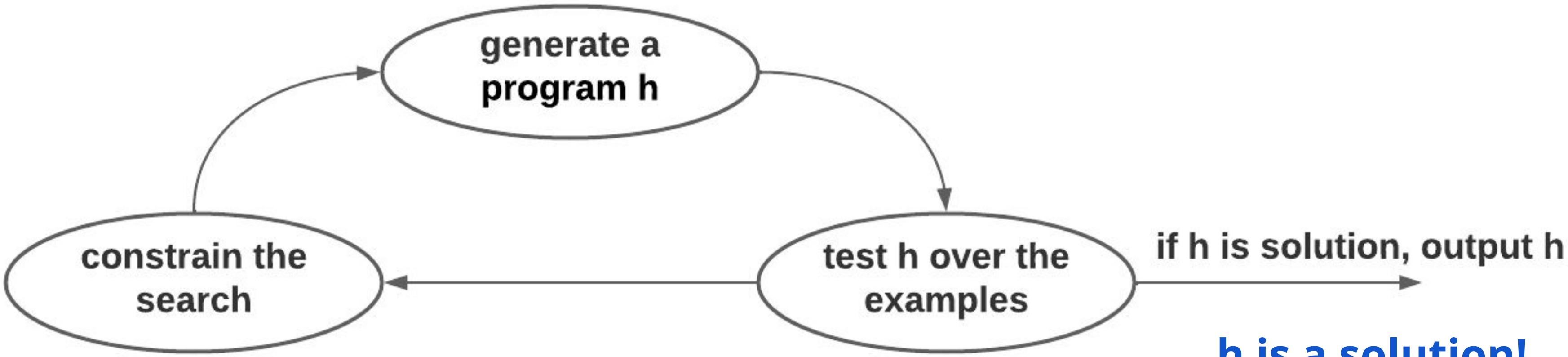
```
zendo(Structure) ← piece(Structure,Piece1),contact(Piece1,Piece2),blue(Piece2).
```



```
zendo(Structure) ← piece(Structure,Piece1),coord(Piece1,X,Y),geq(X,Y).
```



```
zendo(Structure) ← piece(Structure,Piece1),red(Piece1),contact(Piece1,Piece2),square(Piece2).
```



h is a solution!

Theorem: our approach learns an optimal solution (a textually minimal hypothesis) if one exists.

Why does it work?

- We do not precompute the hypothesis space
  - We can handle infinite domains, function symbols (lists)

Why does it work?

- We do not precompute the hypothesis space
  - We can handle infinite domains, function symbols (lists)

- Constraints to prune the hypothesis space

2 - Discovering constraints

Learning logic programs by discovering where not to search, Andrew Cropper and Céline Hocquette, AAAI, 2023

Background Knowledge		
even(0). even(2). even(4). odd(1). odd(3). odd(5). …	succ(0,1). succ(1,2). succ(2,3). succ(3,4). succ(4,5). succ(5,6). …	head([1],1). head([2,3,4],2). head([4,3,2,1],4). head([3],3). head([7,8,9],7). head([6,7,8,9],6). …

odd/1 and even/1 are mutually exclusive

odd/1 and even/1 are mutually exclusive

```
← odd(A), even(A).
```

odd/1 and even/1 are mutually exclusive

```
← odd(A), even(A).
```

```
zendo(A) ← piece(A,B), size(B,C), odd(C), even(C).
zendo(A) ← piece(A,B), blue(B), coord1(B,C), odd(C), even(C).
zendo(A) ← piece(A,B), contact(B,C), coord2(C,D), geq(D,E), odd(E), even(E).
```

succ/2 is irreflexive, injective, functional, antitransitive, antitriangular, and asymmetric.

succ/2 is irreflexive, injective, functional, antitransitive, antitriangular, and asymmetric.

```
← succ(A,A).
← succ(A,B), succ(A,C), B!=C.
← succ(A,B), succ(C,B), C!=A.
← succ(A,B), succ(A,C), C!=A.
← succ(A,B), succ(B,C), succ(A,C).
← succ(A,B), succ(B,C), succ(C,A).
← succ(A,B), succ(B,A).
```

succ/2 is irreflexive, injective, functional, antitransitive, antitriangular, and asymmetric.

```
zendo(A) ← piece(A,B), size(B,C), succ(C,C).
zendo(A) ← piece(A,B), coord2(B,C), coord1(B,D), succ(C,E), succ(D,E).
zendo(A) ← piece(A,B), size(B,C), piece(A,D), size(D,E), succ(C,E), succ(E,C).
zendo(A) ← piece(A,B), coord1(B,C), succ(C,D), succ(D,E), succ(C,E).
zendo(A) ← piece(A,B), coord1(B,C), succ(C,D), succ(D,E), succ(E,C).
```

How does it work?

Name	Property	Constraint	Example
Irreflexive	$\neg p(A,A)$	$\leftarrow p(A,A)$	$\leftarrow brother(A,A)$
Antitransitive	$p(A,B), p(B,C) \rightarrow \neg p(A,C)$	$\leftarrow p(A,B), p(B,C), p(A,C)$	$\leftarrow succ(A,B), succ(B,C), succ(A,C)$
Antitriangular	$p(A,B), p(B,C) \rightarrow \neg p(C,A)$	$\leftarrow p(A,B), p(B,C), p(C,A)$	$\leftarrow tail(A,B), tail(B,C), tail(C,A)$
Injective	$p(A,B), p(C,B) \rightarrow A=C$	$\leftarrow p(A,B), p(C,B), A \neq C$	$\leftarrow succ(A,B), succ(C,B), A \neq C$
Functional	$p(A,B), p(A,C) \rightarrow B=C$	$\leftarrow p(A,B), p(A,C), B \neq C$	$\leftarrow length(A,B), length(A,C), B \neq C$
Asymmetric	$p(A,B) \rightarrow \neg p(B,A)$	$\leftarrow p(A,B), p(B,A)$	$\leftarrow mother(A,B), mother(B,A)$
Exclusive	$p(A) \rightarrow \neg q(A)$	$\leftarrow p(A), q(A)$	$\leftarrow odd(A), even(A)$

We use an ASP program to discover the constraints.
We adopt a closed world assumption.

Why does it work?

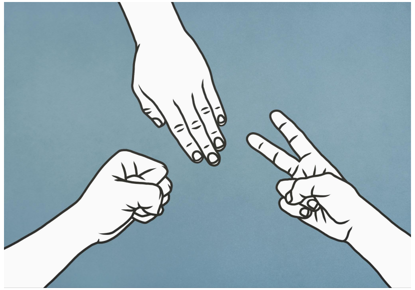
- Only need a counter-example to eliminate a property

Domain	Time
trains	0.22 ± 0.00
zendo	0.03 ± 0.00
imdb	0.02 ± 0.00
krk	0.10 ± 0.00
rps	0.02 ± 0.00
centipede	0.02 ± 0.00
md	0.01 ± 0.00
buttons	0.02 ± 0.00
attrition	0.01 ± 0.00
coins	0.03 ± 0.00
synthesis	4.00 ± 0.40

Background knowledge constraint discovery time (s)

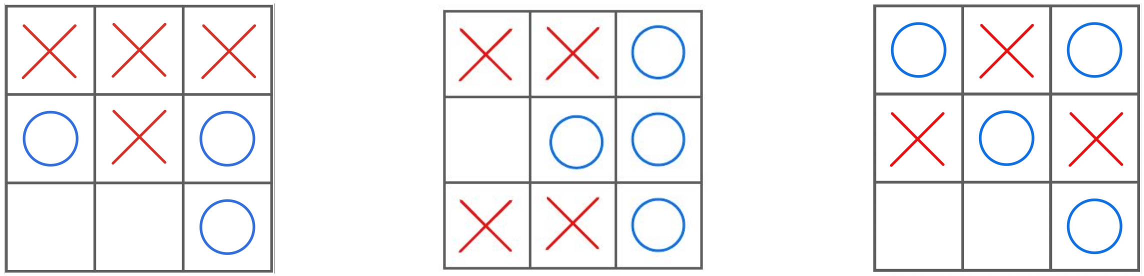
Why does it work?

- Only need a counter-example to eliminate a property
- Constraints can eliminate many hypotheses



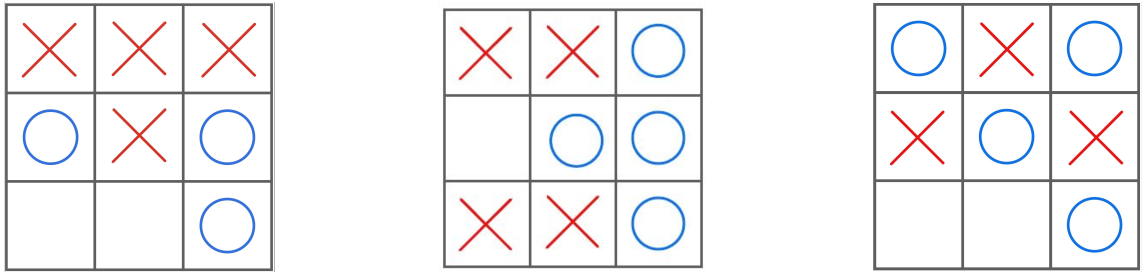
discovering constraints about the succ/2 relation reduces the number of rules in the hypothesis space from 1,189,916 to 70,270, a 94% reduction

3 - Learning programs with many rules



```
win(Board,Player) ← cell(Board,X,0,Player),cell(Board,X,1,Player),cell(Board,X,2,Player)
win(Board,Player) ← cell(Board,0,Y,Player),cell(Board,1,Y,Player),cell(Board,2,Y,Player)
win(Board,Player) ← cell(Board,0,0,Player),cell(Board,1,1,Player),cell(Board,2,2,Player)
win(Board,Player) ← cell(Board,2,0,Player),cell(Board,1,1,Player),cell(Board,0,2,Player)
```

Learning logic programs by combing programs, Andrew Cropper and Céline Hocquette, ECAI, 2023



```
r1: win(Board,Player) ← cell(Board,X,0,Player),cell(Board,X,1,Player),cell(Board,X,2,Player)

r2: win(Board,Player) ← cell(Board,0,Y,Player),cell(Board,1,Y,Player),cell(Board,2,Y,Player)

r3: win(Board,Player) ← cell(Board,0,0,Player),cell(Board,1,1,Player),cell(Board,2,2,Player)

r4: win(Board,Player) ← cell(Board,2,0,Player),cell(Board,1,1,Player),cell(Board,0,2,Player)
```

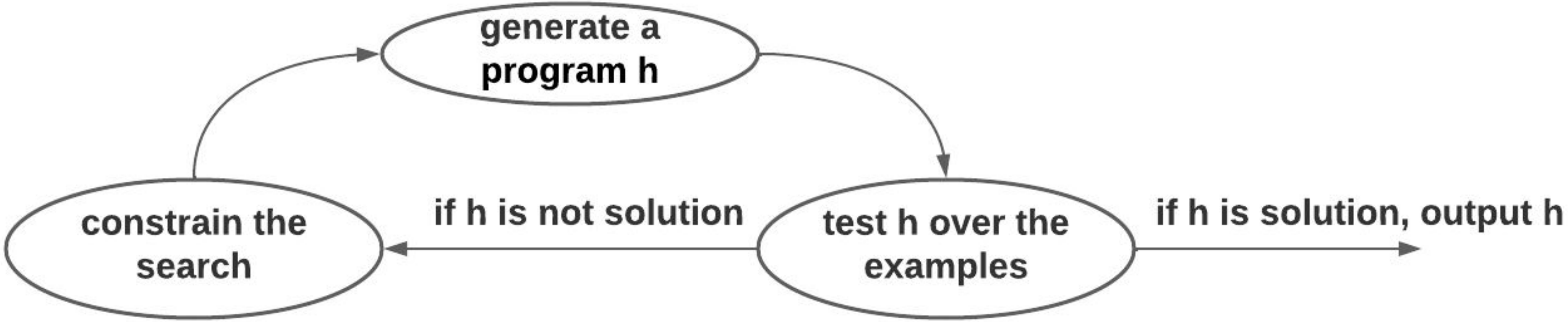
r_1, r_2, r_3 and r_4 do not depend on each other

Idea

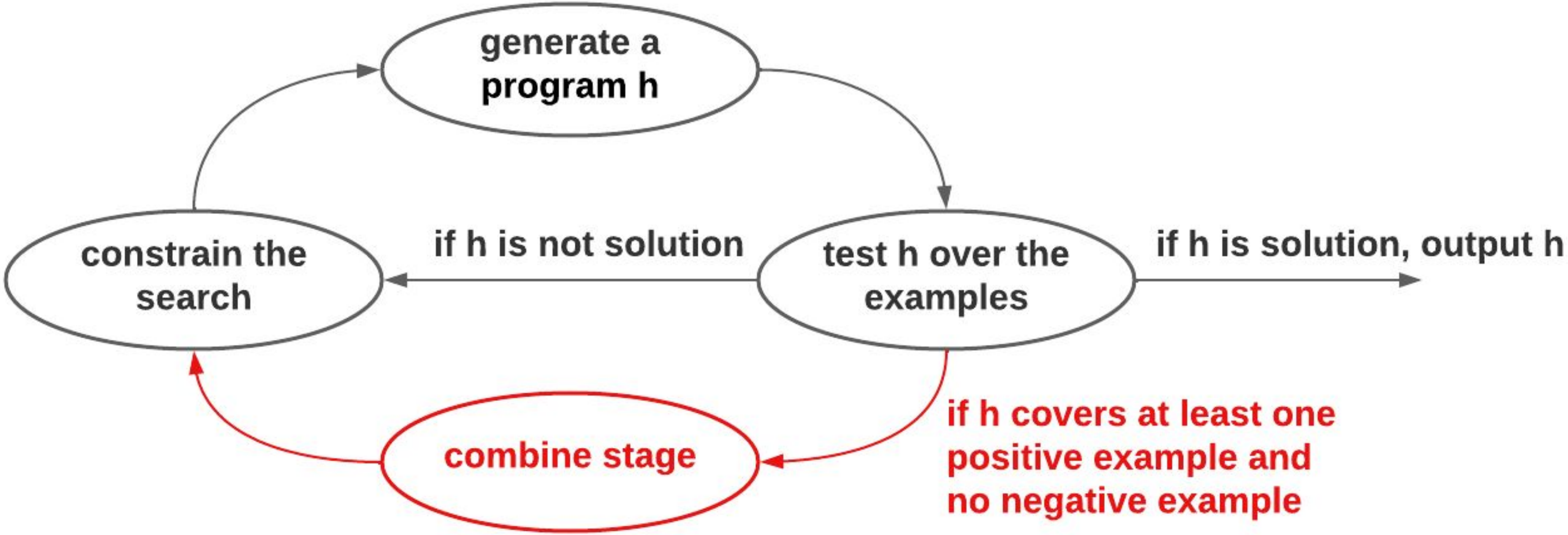
Learn small programs that entail some of the positive examples

Combine these programs to learn programs with many rules that entail many positive examples

Our approach



Our approach



Combine stage

Input: a set P of programs, with their size and coverage, such that for all $p \in P$:

- p covers at least one positive example
- p does not cover any negative example

Combine stage

Input: a set P of programs, with their size and coverage, such that for all $p \in P$:

- p covers at least one positive example
- p does not cover any negative example

Output: a set of programs $P' \subset P$ (a combination of programs) such that:

- P’ covers as many positive examples as possible
- P’ is minimal in size

Combine stage

Input:

Program	Positive examples covered	Size
p1	{e1,e2,e3}	3
p2	{e9}	3
p3	{e1,e3,e5,e6,e7}	4
p4	{e2,e6,e7}	4
p5	{e2,e5,e8,e9}	5
p6	{e8,e9}	6

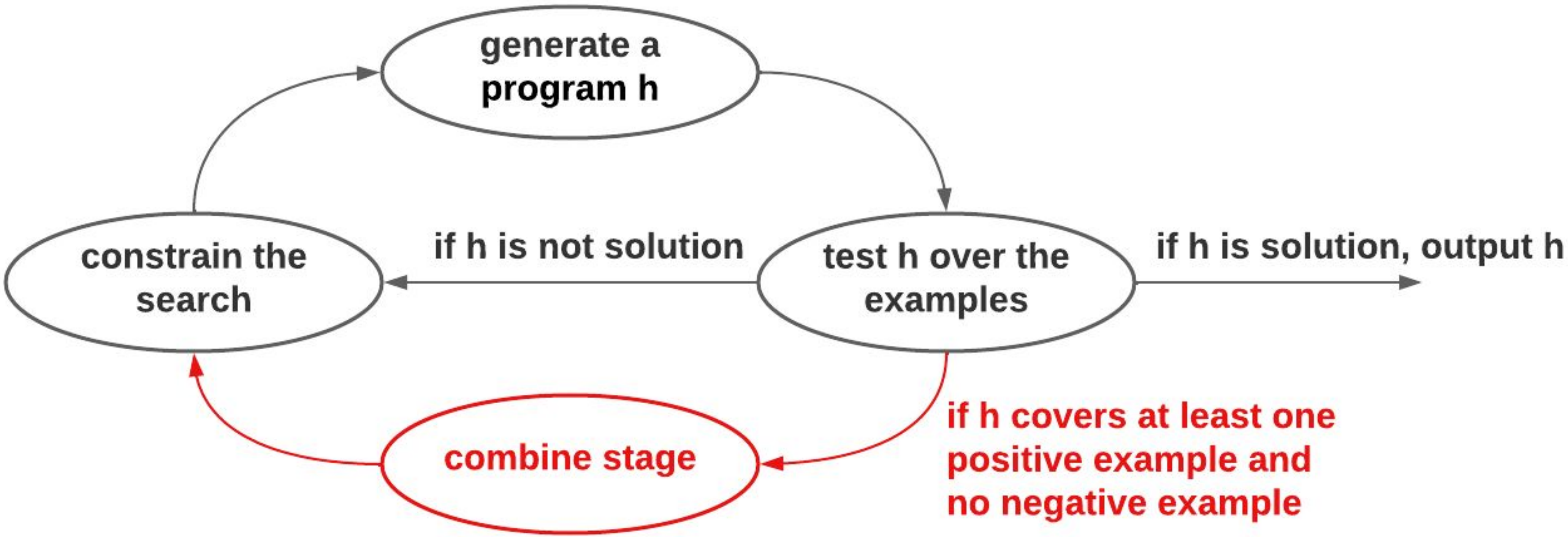
Combine stage

Input:

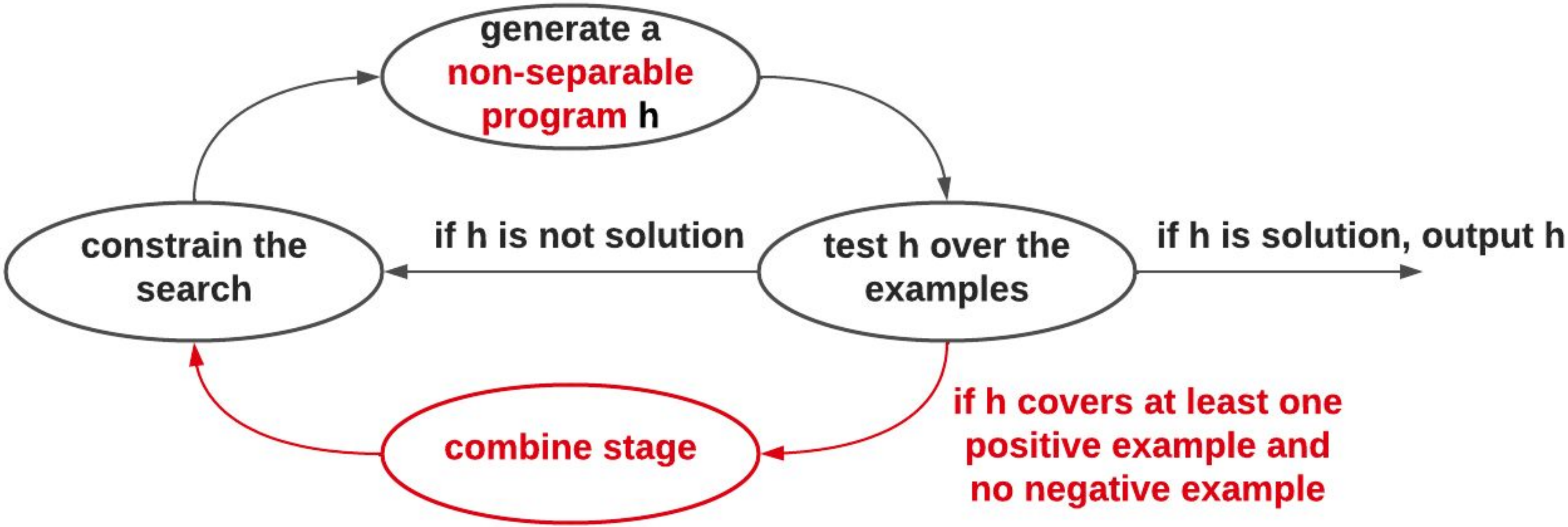
Program	Positive examples covered	Size
p1	{e1,e2,e3}	3
p2	{e9}	3
p3	{e1,e3,e5,e6,e7}	4
p4	{e2,e6,e7}	4
p5	{e2,e5,e8,e9}	5
p6	{e8,e9}	6

Output:
{p1,p3,p5} covers {e1,e2,e3,e5,e6,e7,e8,e9} and has size 12

Our approach



Our approach



```
win(Board,Player) ← cell(Board,X,0,Player),cell(Board,X,1,Player),cell(Board,X,2,Player)

win(Board,Player) ← cell(Board,0,Y,Player),cell(Board,1,Y,Player),cell(Board,2,Y,Player)

win(Board,Player) ← cell(Board,0,0,Player),cell(Board,1,1,Player),cell(Board,2,2,Player)

win(Board,Player) ← cell(Board,2,0,Player),cell(Board,1,1,Player),cell(Board,0,2,Player)
```

```
win(Board,Player) ← cell(Board,X,0,Player),cell(Board,X,1,Player),cell(Board,X,2,Player)

win(Board,Player) ← cell(Board,0,Y,Player),cell(Board,1,Y,Player),cell(Board,2,Y,Player)

win(Board,Player) ← cell(Board,0,0,Player),cell(Board,1,1,Player),cell(Board,2,2,Player)

win(Board,Player) ← cell(Board,2,0,Player),cell(Board,1,1,Player),cell(Board,0,2,Player)
```

Separable program

```
line(Board,0,Player) ← cell(Board,0,Player)
line(Board,Cell,Player) ← cell(Board,Cell,Player), above(Cell,Cell1), line(Board,Cell1,Player)
```

```
line(Board,0,Player) ← cell(Board,0,Player)
line(Board,Cell,Player) ← cell(Board,Cell,Player), above(Cell,Cell1), line(Board,Cell1,Player)
```

Non-separable program

Why does it work?

- Searching over non-separable programs only can vastly reduce the hypothesis space.

Why does it work?

- Searching over non-separable programs only can vastly reduce the hypothesis space.

m rules in the hypothesis space,
at most k rules in a program

separable	non-separable
m^k	m

Why does it work?

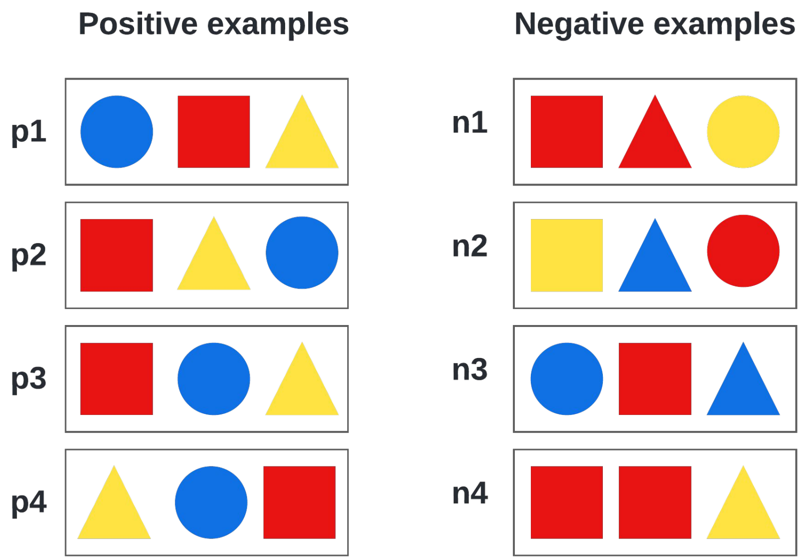
- Searching over non-separable programs only can vastly reduce the hypothesis space.
- We can leverage recent progress in solvers

Task	combine	no combine
zendo1	3 ± 1	7 ± 1
zendo2	49 ± 5	*timeout*
zendo3	55 ± 6	*timeout*
zendo4	53 ± 11	3243 ± 359

Table 1: Learning times (seconds) with a 60 minutes timeout

Theorem: our approach learns an optimal solution (a textually minimal hypothesis) if one exists.

4 - Learning programs with big rules




```
zendo(Structure) ←
    piece(Structure,Piece1),blue(Piece1),round(Piece1),
    piece(Structure,Piece2),red(Piece2),square(Piece2),
    piece(Structure,Piece3),yellow(Piece3),triangle(Piece3)
.
```

Learning big logical rules by joining small rules, Céline Hocquette, Andreas Niskanen, Rolf Morel, Matti Järvisalo, and Andrew Cropper, IJCAI, 2024.

Idea

Learn small rules that entail some positive and some negative examples

```
zendo1(Structure) ← piece(Structure,Piece1),blue(Piece1),round(Piece1).

zendo2(Structure) ← piece(Structure,Piece2),red(Piece2),square(Piece2).

zendo3(Structure) ← piece(Structure,Piece3),yellow(Piece3),triangle(Piece3).
```

Idea

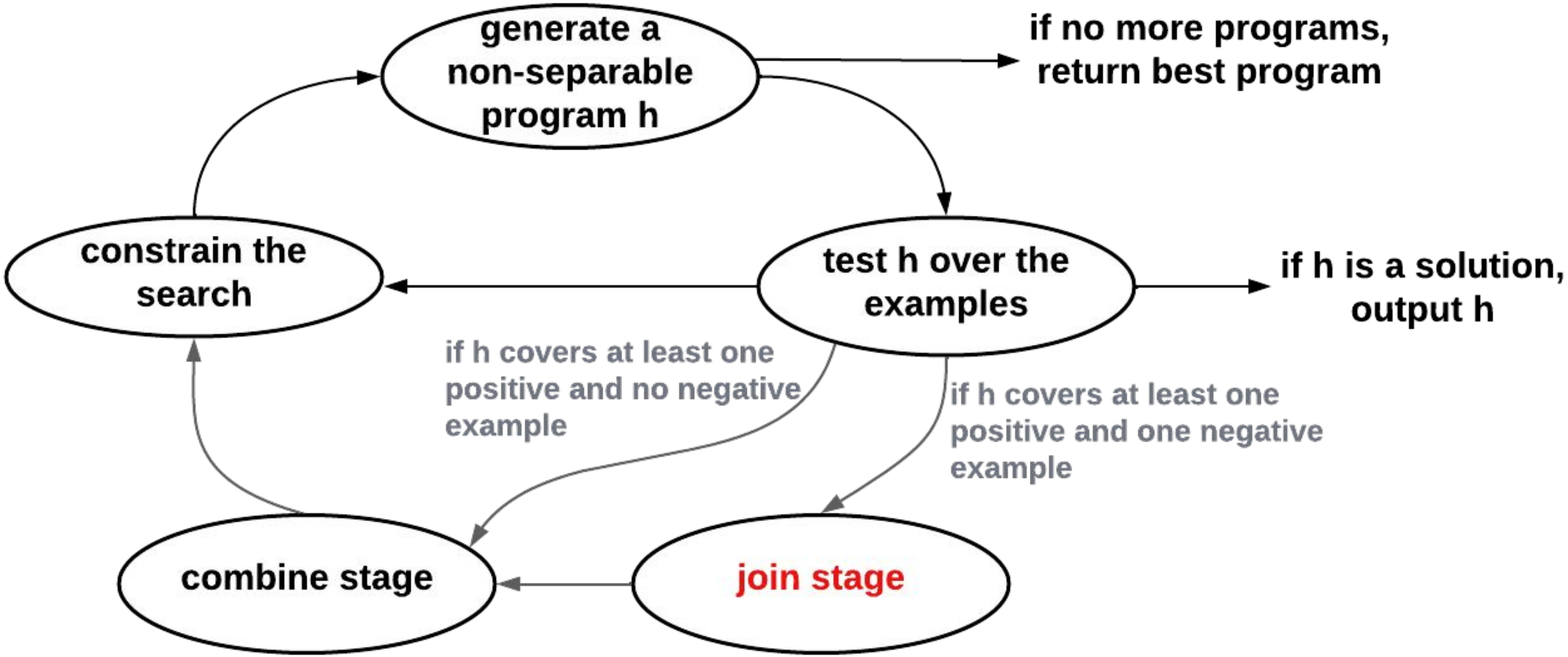
Learn small rules that entail some positive and some negative examples

```
zendo1(Structure) ← piece(Structure,Piece1),blue(Piece1),round(Piece1).

zendo2(Structure) ← piece(Structure,Piece2),red(Piece2),square(Piece2).

zendo3(Structure) ← piece(Structure,Piece3),yellow(Piece3),triangle(Piece3).
```

Join these rules to learn big rules that entail some positive examples and no negative examples

```
zendo1(Structure) ← zendo1(Structure),zendo2(Structure),zendo3(Structure).
```

Our approach



Join stage

Input: a set P of programs, with their size and coverage, such that for all $p \in P$:

- p covers at least one positive example
- p covers at least one negative example

Join stage

Input: a set P of programs, with their size and coverage, such that for all $p \in P$:

- p covers at least one positive example
- p covers at least one negative example

Output: sets of programs $P' \subset P$ (conjunctions of programs) such that:

- P' does not cover any negative example

Join stage

Input:

Program	Positive examples covered	Negative examples covered	Size
p1	{e1}	{n3}	2
p2	{e2}	{n3}	2
p3	{e1,e2}	{n1,n2}	3
p4	{e1,e2}	{n1,n3}	5
p5	{e1,e2}	{n1,n2}	5

Join stage

Input:

Program	Positive examples covered	Negative examples covered	Size
p1	{e1}	{n3}	2
p2	{e2}	{n3}	2
p3	{e1,e2}	{n1,n2}	3
p4	{e1,e2}	{n1,n3}	5
p5	{e1,e2}	{n2,n3}	5

Output:
c1={p3,p4,p5} covers {e1,e2} and has size 13

Join stage

Input:

Program	Positive examples covered	Negative examples covered	Size
p1	{e1}	{n3}	2
p2	{e2}	{n3}	2
p3	{e1,e2}	{n1,n2}	3
p4	{e1,e2}	{n1,n3}	5
p5	{e1,e2}	{n1,n2}	5

Output:
c1={p3,p4,p5} covers {e1,e2} and has size 13
c2={p1,p3} covers {e1} and has size 5

Join stage

Input:

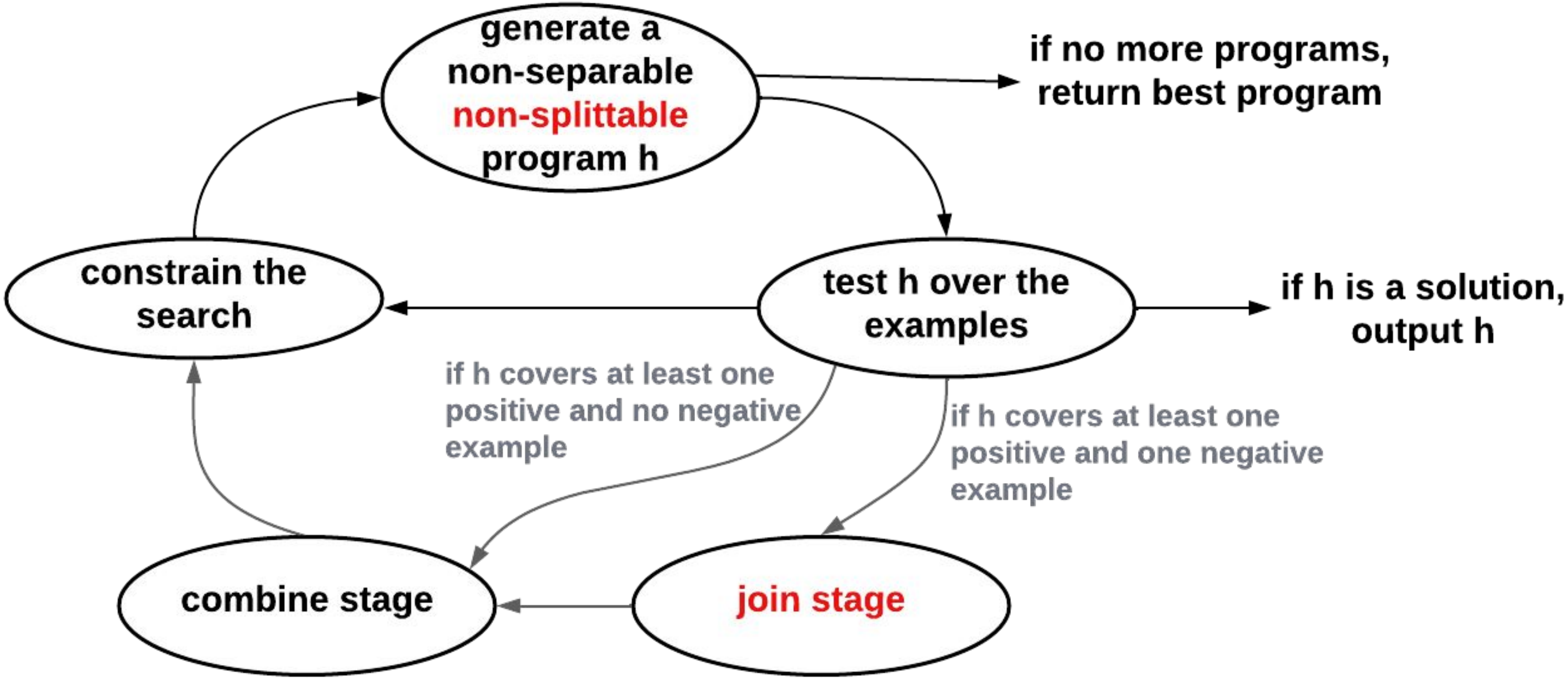
Program	Positive examples covered	Negative examples covered	Size
p1	{e1}	{n3}	2
p2	{e2}	{n3}	2
p3	{e1,e2}	{n1,n2}	3
p4	{e1,e2}	{n1,n3}	5
p5	{e1,e2}	{n1,n2}	5

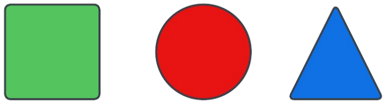
Output:

c1={p3,p4,p5} covers {e1,e2} and has size 13

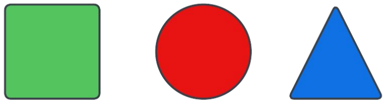
c2={p1,p3} covers {e1} and has size 5

c3={p2,p3} covers {e2} and has size 5


generate a
non-separable
non-splittable
program h
if no more programs,
return best program
constrain the
search
test h over the
examples
if h is a solution,
output h
if h covers at least one
positive and no negative
example
if h covers at least one
positive and one negative
example
combine stage
join stage



```
zendo(Structure) ← piece(Structure,Piece1), blue(Piece1), triangle(Piece1),
                   piece(Structure,Piece2), square(Piece2), left(Piece2,Piece3), red(Piece3)
```

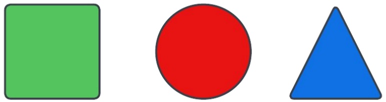


Head variable

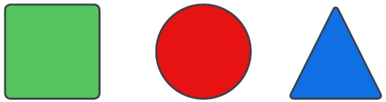
body-only variable

```
zendo(Structure) ← piece(Structure,Piece1), blue(Piece1), triangle(Piece1),
                   piece(Structure,Piece2), square(Piece2), left(Piece2,Piece3), red(Piece3)
```

Splittable program



```
zendo(Structure) ← piece(Structure,Piece1), blue(Piece1), triangle(Piece1),
                   piece(Structure,Piece2), square(Piece2), left(Piece2,Piece3), red(Piece3)
                   left(Piece1,Piece2)
```



```
zendo(Structure) ← piece(Structure,Piece1), blue(Piece1), triangle(Piece1),
                   piece(Structure,Piece2), square(Piece2), left(Piece2,Piece3), red(Piece3)
                   left(Piece1,Piece2)
```

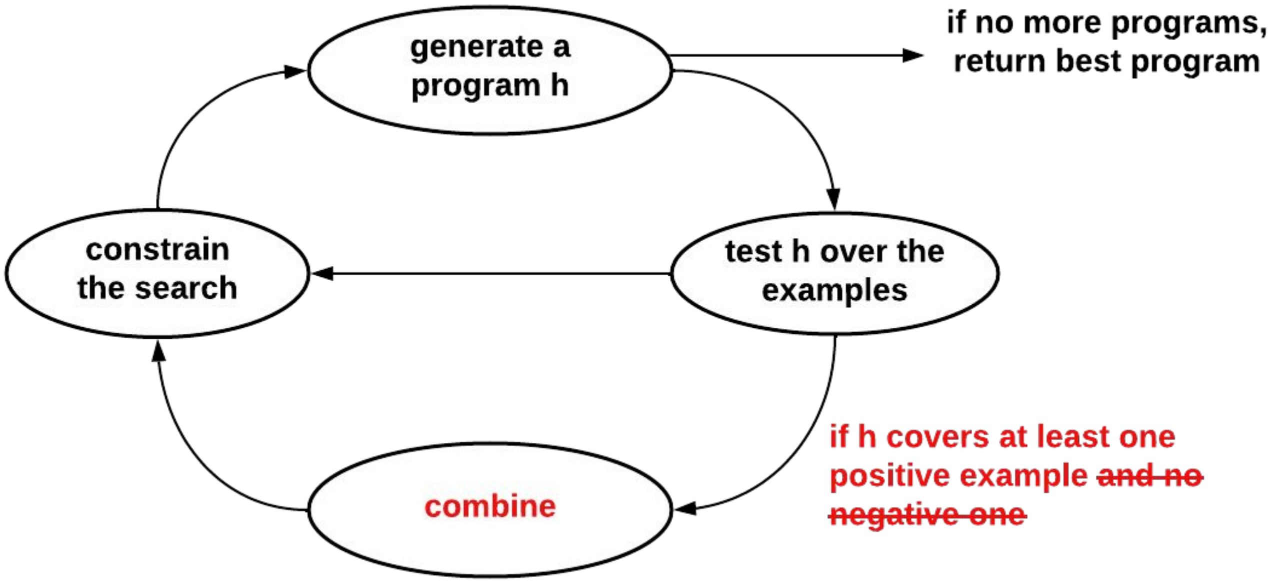
Non-splittable program

Why does it work?

- Searching over non-splittable programs only can vastly reduce the hypothesis space.

- We can leverage recent progress in SAT-solvers

Future projects: which cost function?



We use a MaxSAT solver to search for an optimal combination of programs

Which cost function?

- minimum description length: trade-off model complexity (program size) and data fit (training accuracy)

Learning MDL logic programs from noisy data, Céline Hocquette, Andreas Niskanen, Matti Järvisalo, and Andrew Cropper, AAAI, 2024.

Which cost function?

- minimum description length: trade-off model complexity (program size) and data fit (training accuracy)
- is minimising the size of programs important?
- learning from positive only data

Thank you!

celinehocquette@gmail.com

Questions?