

Artificial Intelligence and Domain-Specific Languages for Inductive Theorem Proving

PhD defence of Yutaka Nagashima, 17.06.2021



Why theorem proving?



- ✓ Why theorem proving?
- ✓ What is Isabelle/HOL?



- ✓ Why theorem proving?
- ✓ What is Isabelle/HOL?
- ✓ What are tactics?



- ✓ Why theorem proving?
- ✓ What is Isabelle/HOL?
- ✓ What are tactics?

The screenshot shows the Isabelle/HOL proof assistant interface. The top bar includes standard file operations like Open, Save, and Undo. The main window displays a theory file named "Kaiserslautern.thy". The code defines three entries:

- A primitive recursive function `rev1` that takes a list and returns its reverse. It is defined by two cases: an empty list which is itself, and a non-empty list where the head is prepended to the reverse of the tail.
- A function `rev2` that also takes a list and returns its reverse. It is defined by two cases: an empty list which is itself, and a non-empty list where the head is prepended to the reverse of the tail.
- A theorem stating that `rev2 xs ys = rev1 xs @ ys`. This is proved using the `induct` tactic on `xs ys` and the `rule: rev2.induct`.

```
File Browser Documentation
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
  apply auto done
```



- ✓ Why theorem proving?
- ✓ What is Isabelle/HOL?
- ✓ What are tactics?



- ✓ Why theorem proving?
- ✓ What is Isabelle/HOL?
- ✓ What are tactics?
- ✓ Why proof by induction?



- ✓ Why theorem proving?
- ✓ What is Isabelle/HOL?
- ✓ What are tactics?
- ✓ Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

Strategic Issues, Problems and Challenges in Inductive Theorem Proving



ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- ✓ What is Isabelle/HOL?
- ✓ What are tactics?
- ✓ Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

Strategic Issues, Problems and Challenges in Inductive Theorem Proving



ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.



What is Isabelle/HOL?



What are tactics?



Why proof by induction?



we are convinced that substantial progress in ITP will take time.

Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

Strategic Issues, Problems and Challenges in Inductive Theorem Proving



ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

What is Isabelle/HOL?



What are tactics?



Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving



ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.



What is Isabelle/HOL?



What are tactics?



Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

When applying AI for theorem proving, keep in mind...





ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

What is Isabelle/HOL?

What are tactics?

Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

When applying AI for theorem proving, keep in mind...

Algorithmic approaches work well, but not always.





ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.



What is Isabelle/HOL?



What are tactics?



Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

When applying AI for theorem proving, keep in mind...

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.





ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.



What is Isabelle/HOL?



What are tactics?



Why proof by induction?

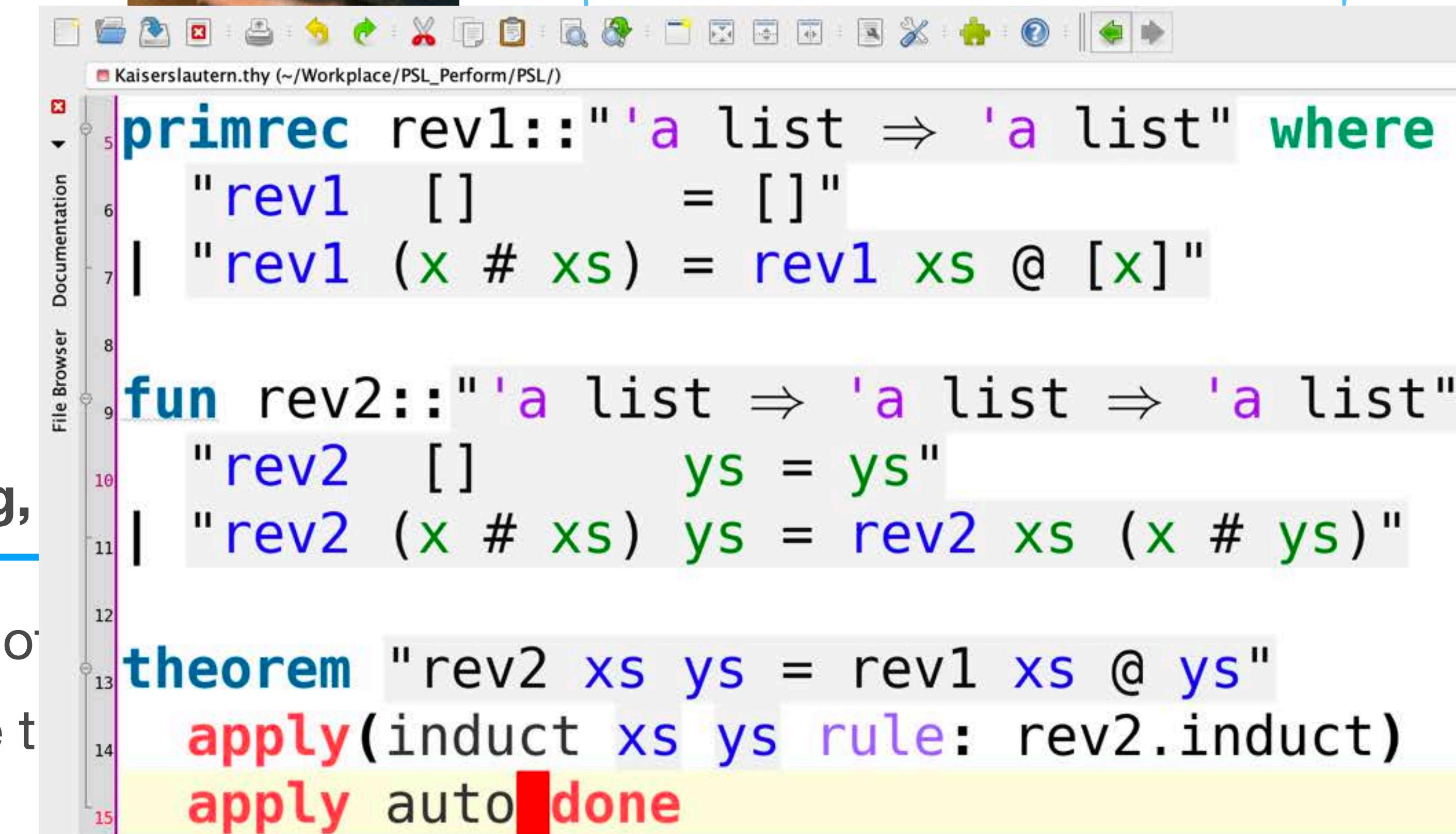
When applying AI for theorem proving,

Algorithmic approaches work well, but no

There are many good ways to prove one t



we are convinced that substantial progress in ITP will take time.



```
File Browser Documentation
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list"
  "rev2 []      ys = ys"
  "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
  apply auto done
```

proof (prove)
goal:
No subgoals!

Proof state Auto up



ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.



What is Isabelle/HOL?



What are tactics?



Why proof by induction?

When applying AI for theorem proving,

Algorithmic approaches work well, but no

There are many good ways to prove one t

```
| "rev2 (x # xs) ys = rev2 xs (x # ys)"  
|  
11  
12  
13 theorem "rev2 xs ys = rev1 xs @ ys"  
14 apply(induct xs arbitrary: ys)  
15 apply auto done  
  
proof (prove)  
goal:  
No subgoals!
```

```
x ▾ Output Query Sledgehammer Symbols  
15.13 (344/353)  
13 theorem "rev2 xs ys = rev1 xs @ ys"  
14 apply(induct xs ys rule: rev2.induct)  
15 apply auto done  
  
proof (prove)  
goal:  
No subgoals!
```



ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

What is Isabelle/HOL?

What are tactics?

Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

When applying AI for theorem proving, keep in mind...

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.





ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.



What is Isabelle/HOL?



What are tactics?



Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

When applying AI for theorem proving, keep in mind...

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.





ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.



What is Isabelle/HOL?



What are tactics?



Why proof by induction?



Prof. Bernhard Gramlich
<https://www.logic.at/staff/gramlich/>

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

When applying AI for theorem proving, keep in mind...

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



Path to automatic induction



Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).



Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.



Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).



Path to automatic induction

Step 1: Tactic recommendation (PaMPeR).

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).

b: Logical Feature Extraction (LiFtEr).

c: smart_induct using LiFtEr.



Path to automatic induction

Step 1: Tactic recommendation (PaMPeR).

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).

- b: Logical Feature Extraction (LiFtEr).

- c: smart_induct using LiFtEr.

Step 3: Putting things together.



Step1: Proof Method Recommendation (PaMpeR)

The screenshot shows the PaMpeR interface with a proof state. The state contains the following code:

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  | "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
```

The line "theorem" is highlighted with a yellow background. The word "oops" is also present in the code.

```
proof (prove)
goal (1 subgoal):
  1. rev2 xs ys = rev1 xs @ ys
```

Step1: Proof Method Recommendation (PaMpeR)

The screenshot shows the PaMpeR interface with the following code:

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
  "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  which_method
```

The code defines two functions, `rev1` and `rev2`, and a theorem. The `rev1` function is a primitive recursive definition that reverses a list by concatenating the head with the reverse of the tail. The `rev2` function is a standard recursive definition that concatenates the head with the result of reversing the tail. The theorem states that `rev2` is equivalent to `rev1`. The `which_method` command is highlighted in red, indicating it is the current step being recommended.

Step1: Proof Method Recommendation (PaMpeR)

The screenshot shows the PaMpeR interface with a proof script in the central editor area. The script defines three entries: a primitive recursive function `rev1`, a regular function `rev2`, and a theorem. The `which_method` command is highlighted with a red rectangle.

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
  "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  which_method
```

At the bottom of the interface, there are several status indicators: Proof state, Auto update, Update, Search: (empty), and a zoom level of 100%.

Promising methods for this proof goal are:

- simp with expectation of 0.411909082795
- auto with expectation of 0.159332679097
- rule with expectation of 0.0874798467373
- induction with expectation of 0.0613706657985
- metis with expectation of 0.052603838226
- induct with expectation of 0.0510162518838

Step1: Proof Method Recommendation (PaMpeR)

The screenshot shows a software interface for formal verification or proof assistants. The main window displays a proof script in a domain-specific language (PSL). The script includes definitions of two functions, `rev1` and `rev2`, and a theorem. The `which_method` command is highlighted with a red rectangle.

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  | "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  which_method
```

At the bottom of the interface, there are several status indicators: Proof state, Auto update, Update, Search: (empty), and a zoom level of 100%.

Promising methods for this proof goal are:

simp with expectation of 0.411909082795

auto with expectation of 0.159332679097

rule with expectation of 0.0874798467373

induction with expectation of 0.0613706657985

metis with expectation of 0.052603838226

induct with expectation of 0.0510162518838

Step1: Proof Method Recommendation (PaMpeR)



```
primrec rev1::"a list ⇒ 'a list" where
  "rev1 []      = []"
  "rev1 (x # xs) = rev1 xs @ [x]"
```

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



which_method

14 Proof state Auto update Update Search: 100%

Promising methods for this proof goal are:

simp with expectation of 0.411909082795

auto with expectation of 0.159332679097

rule with expectation of 0.0874798467373

induction with expectation of 0.0613706657985

metis with expectation of 0.052603838226

induct with expectation of 0.0510162518838

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

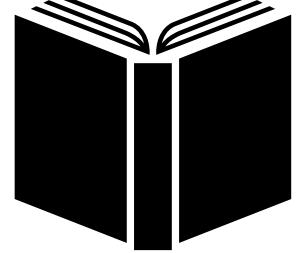
**How does
PaMpeR work?**

recommendation phase

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

large proof corpora



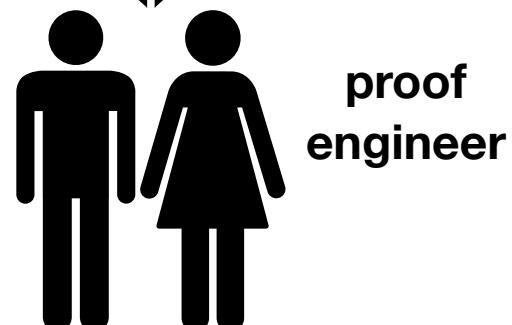
AFP and standard library

How does
PaMpeR work?

recommendation phase



proof
state



proof
engineer

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

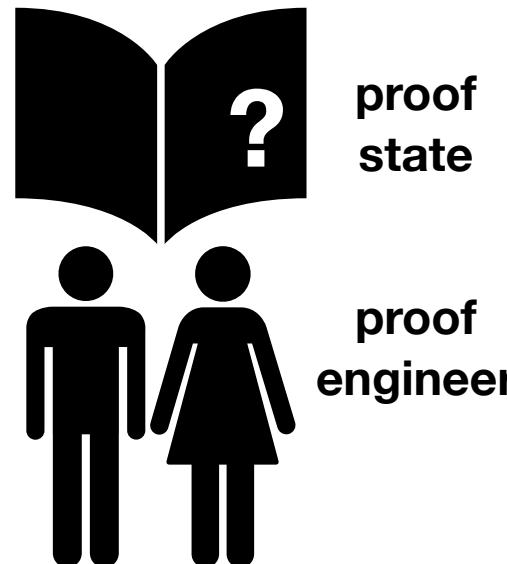
large proof corpora



AFP and standard library

How does
PaMpeR work?

recommendation phase



Archive of Formal Proofs (<https://www.isa-afp.org>)

STATISTICS

Number of Articles: 468
Number of Authors: 313
Number of lemmas: ~128,900
Lines of Code: ~2,170,300

Statistics

Most used AFP articles:

| Name | Used by ? articles |
|------------------------------|--------------------|
| Collections | 15 |
| List-Index | 14 |
| Coinductive | 12 |
| Regular-Sets | 12 |

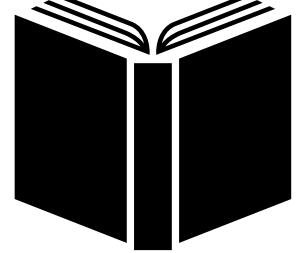
Isabelle logo

Home
About
Submission
Updating Entries
Using Entries
Search
Statistics

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

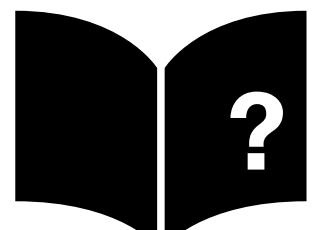
large proof corpora



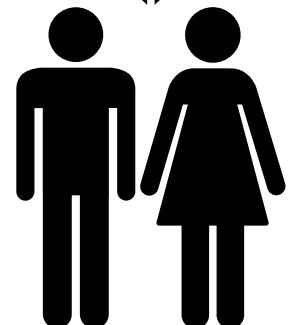
AFP and standard library

How does
PaMpeR work?

recommendation phase

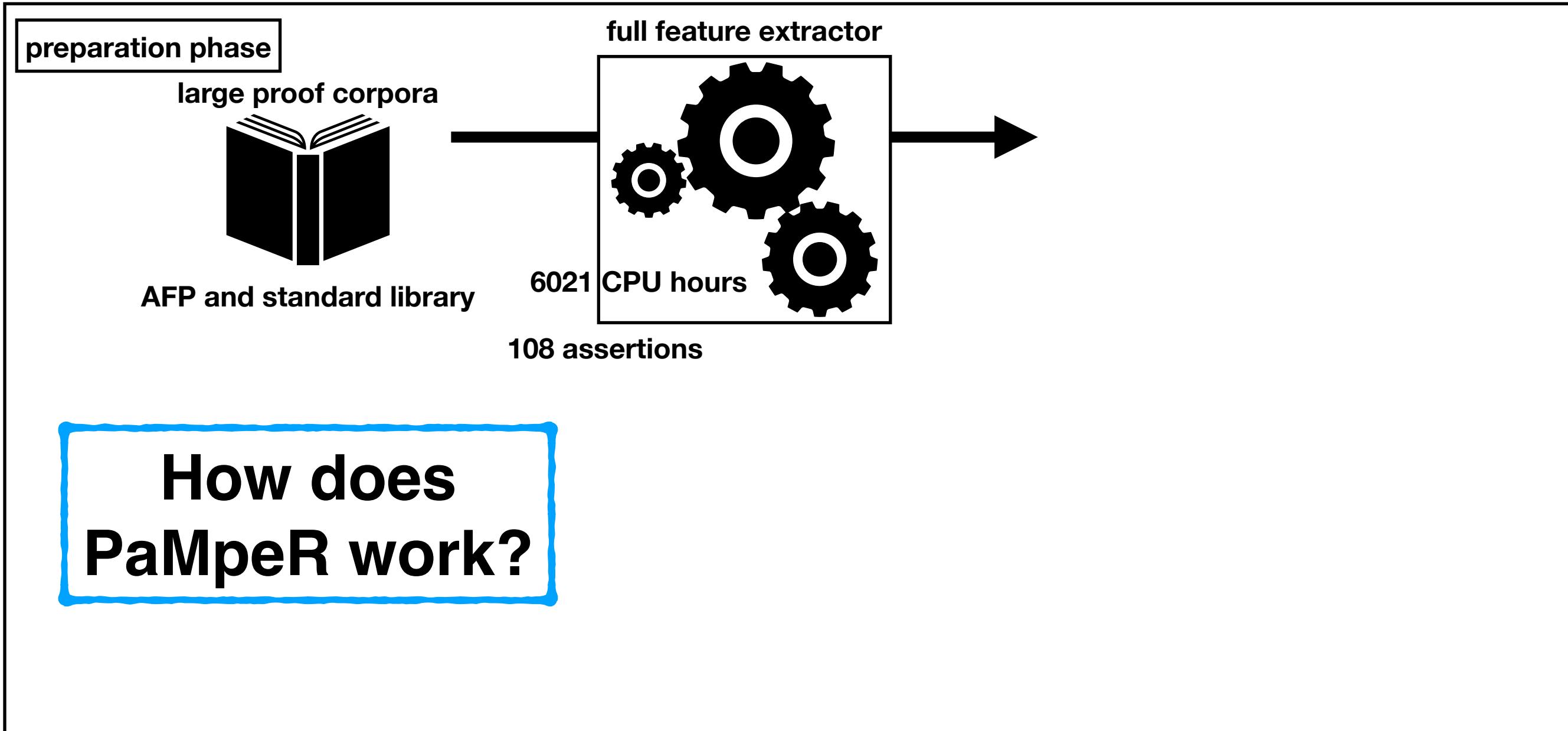


proof
state

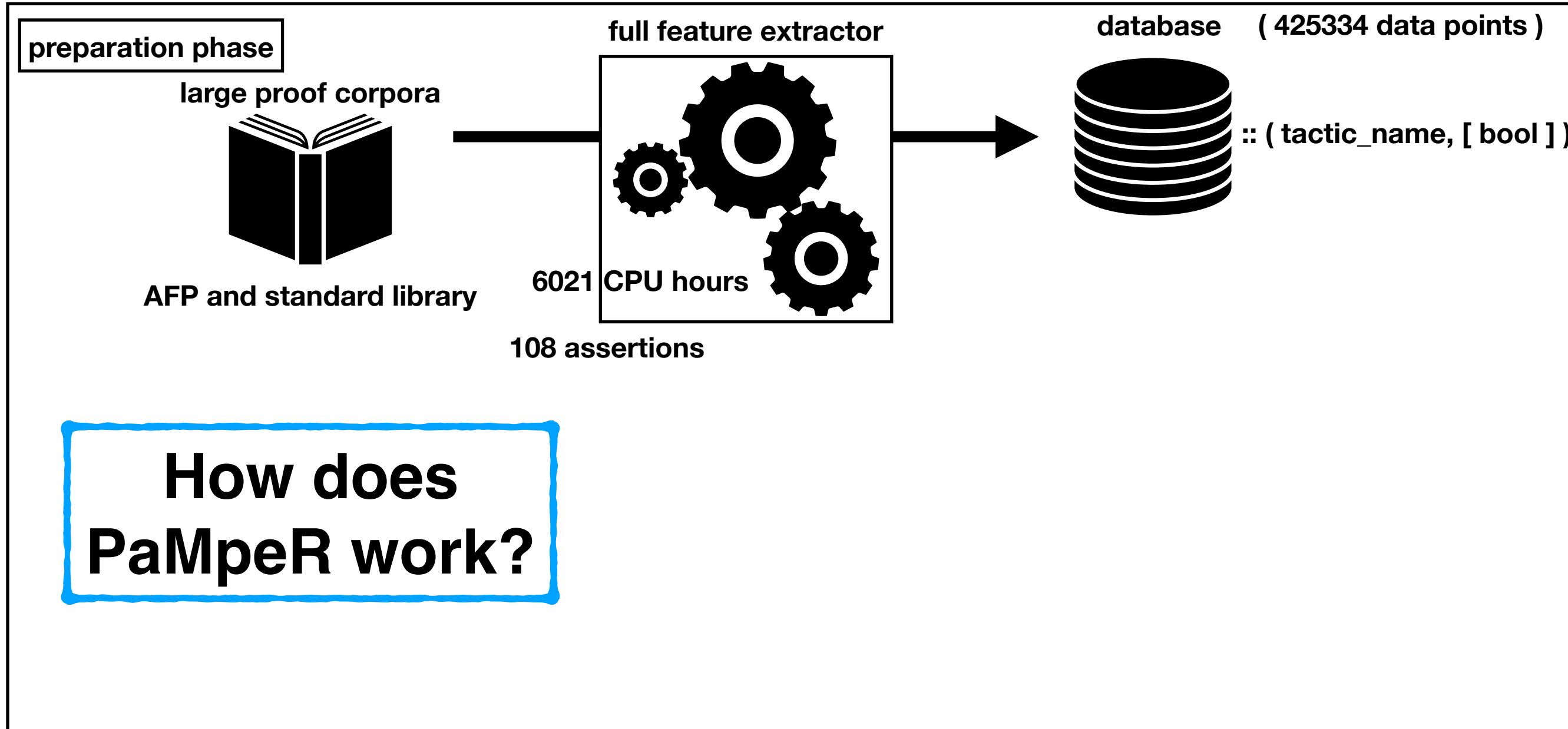


proof
engineer

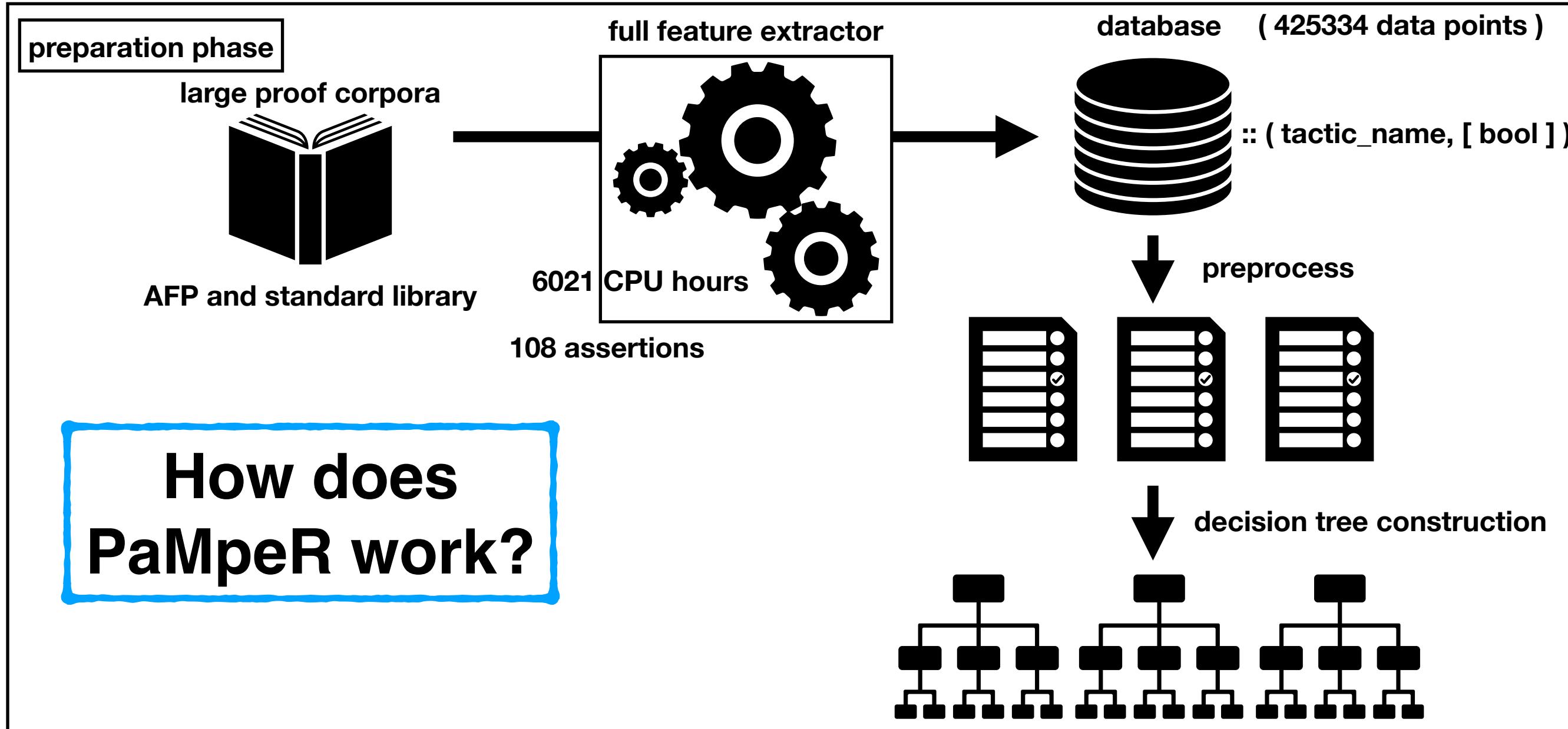
Step1: Proof Method Recommendation (PaMpeR)



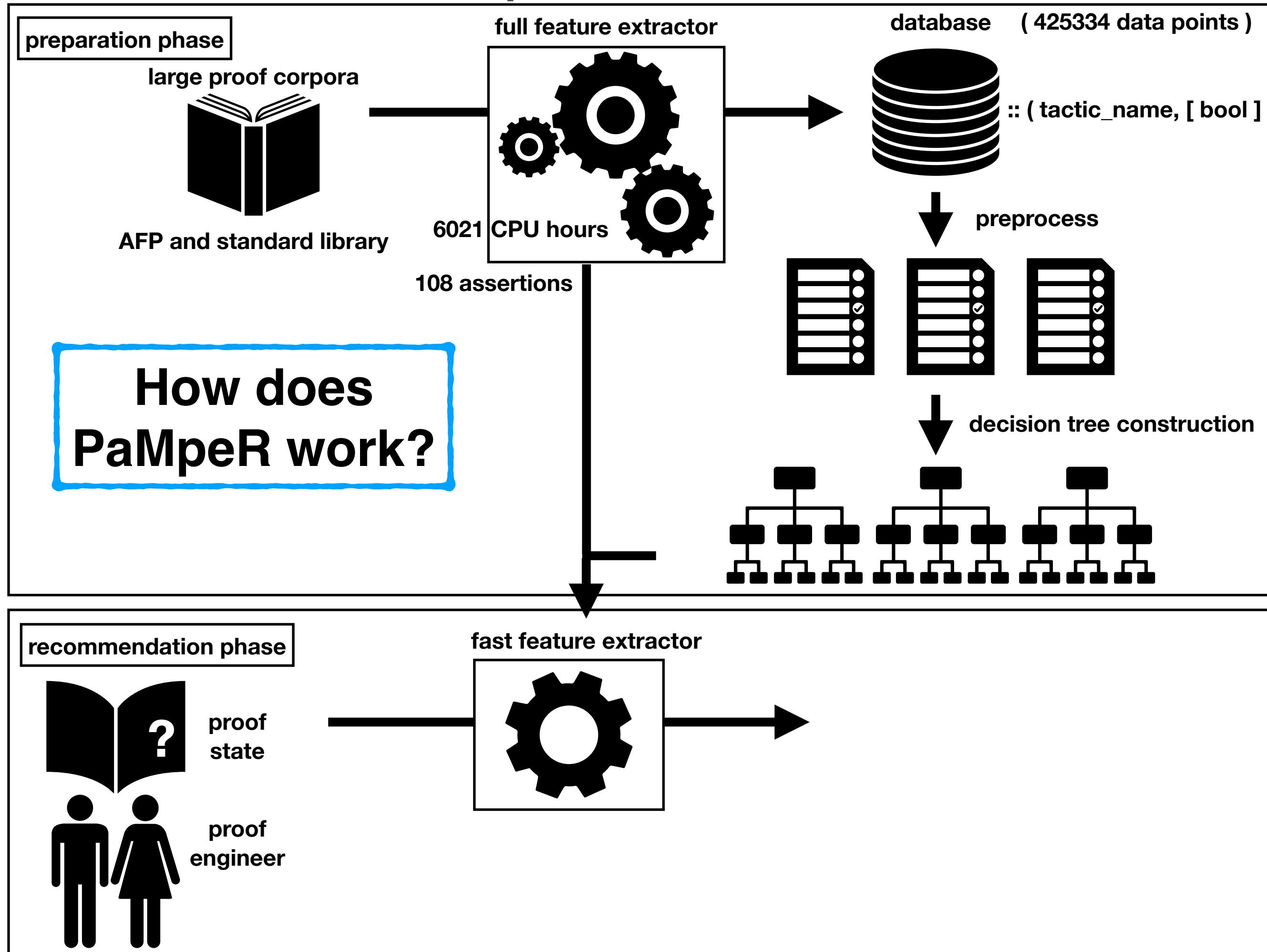
Step1: Proof Method Recommendation (PaMpeR)



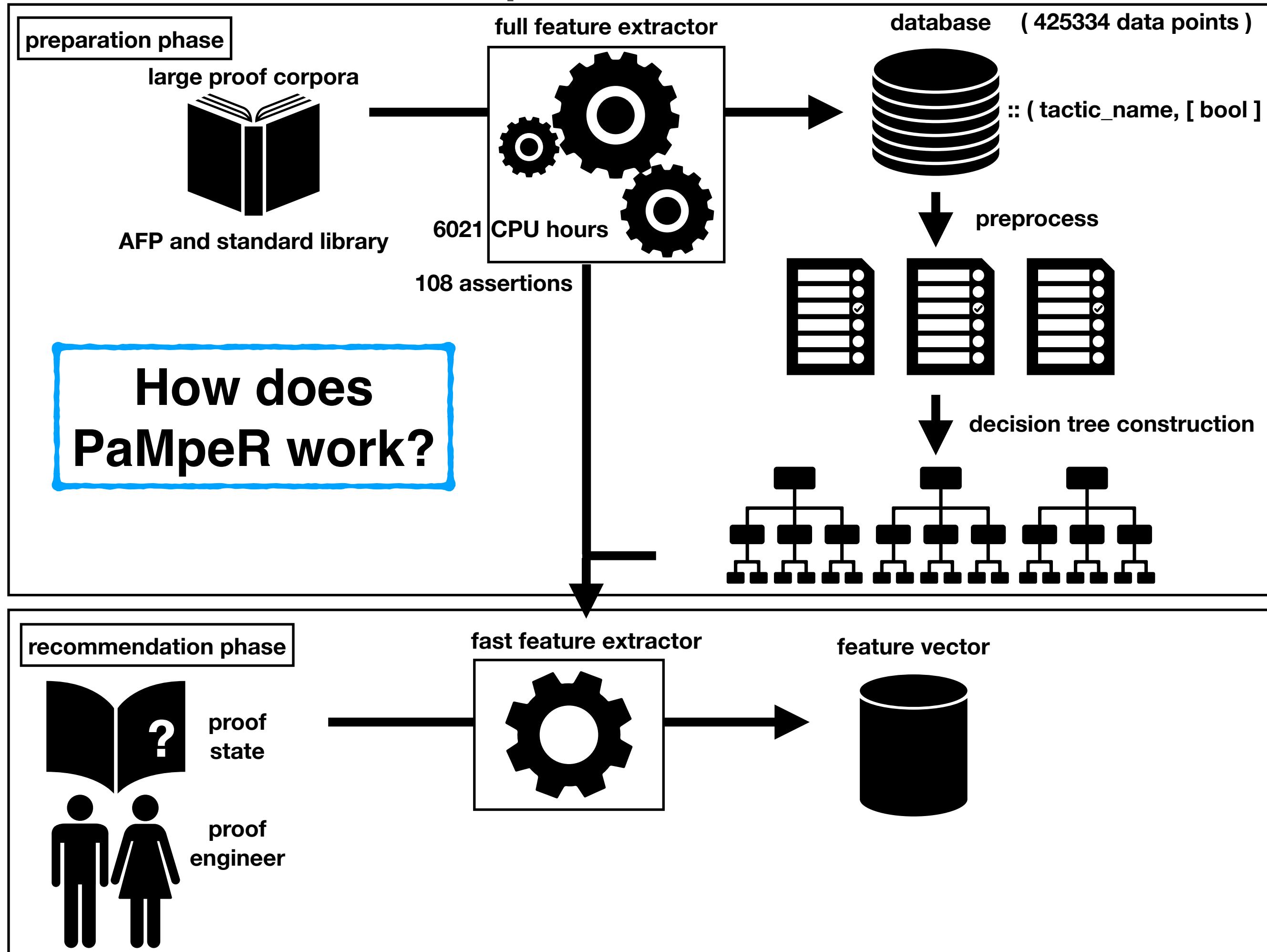
Step1: Proof Method Recommendation (PaMpeR)



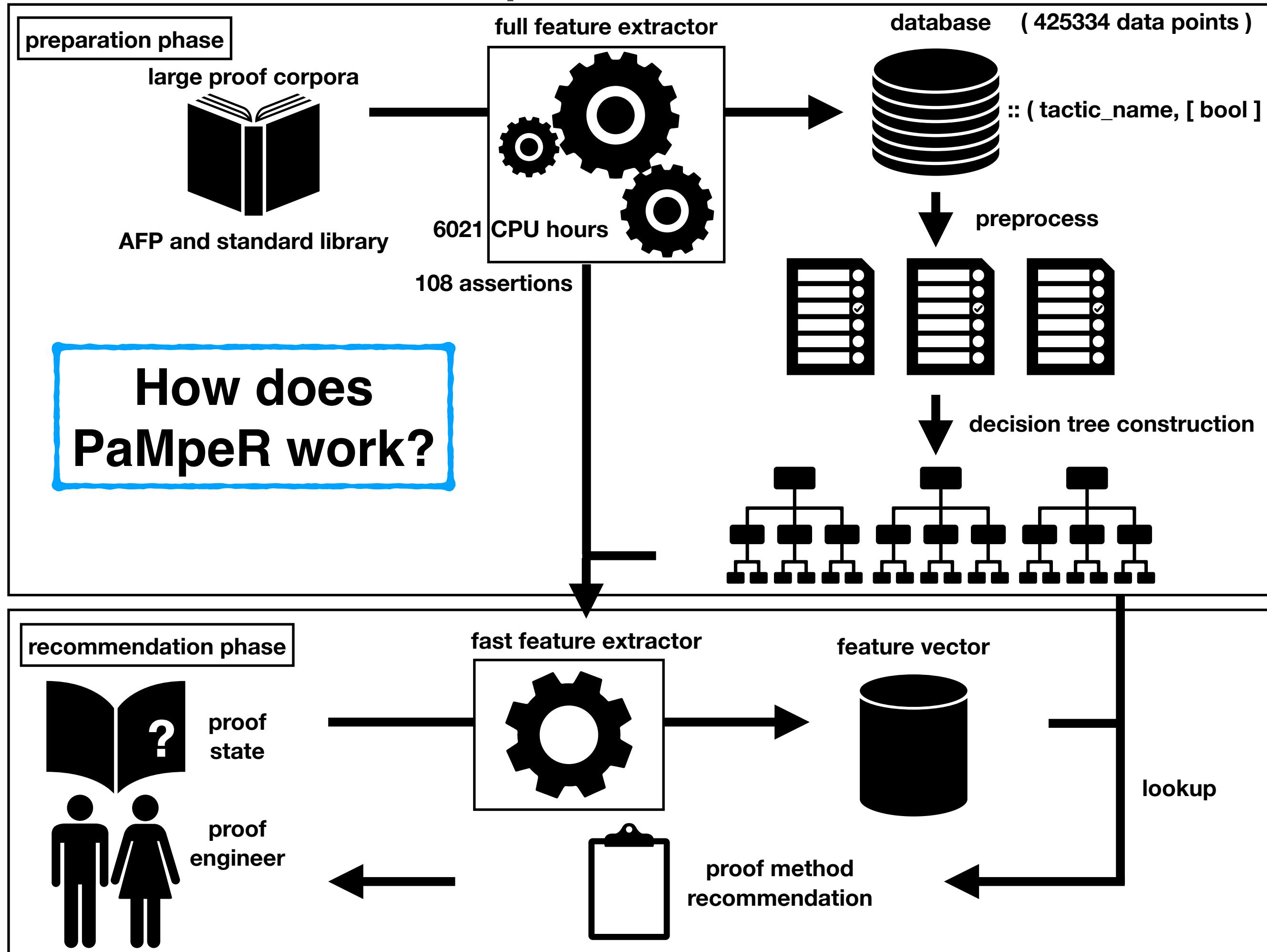
Step1: Proof Method Recommendation (PaMpeR)



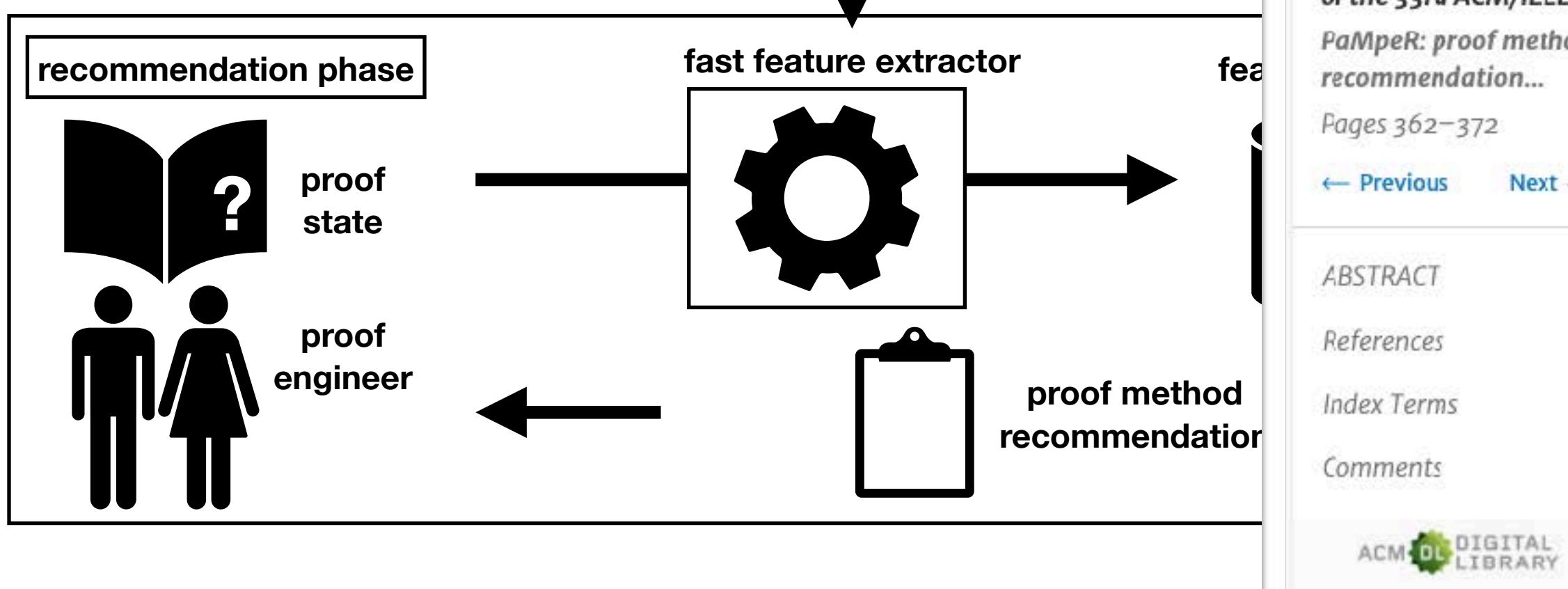
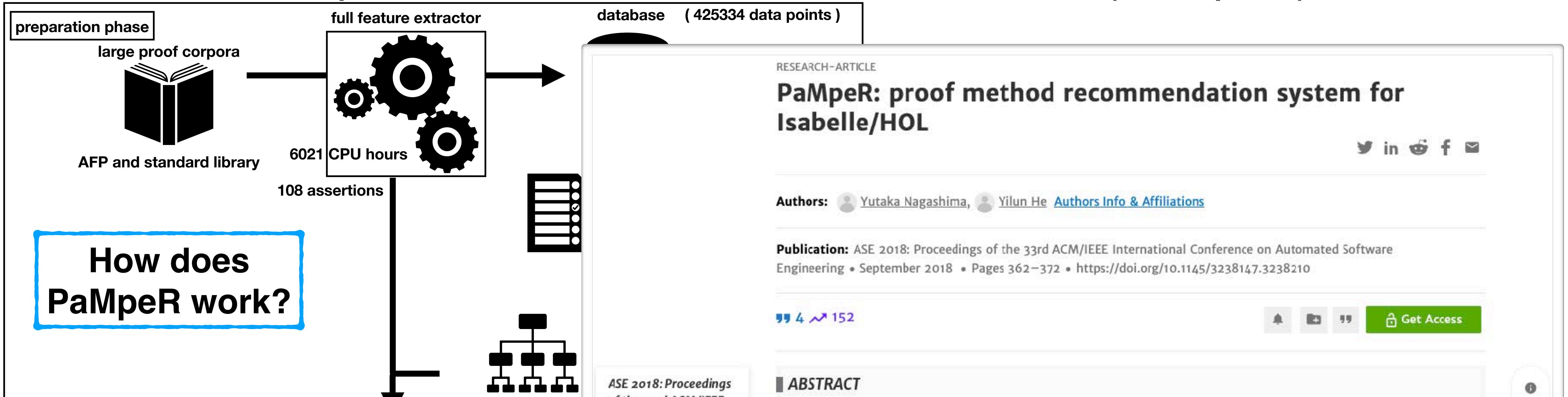
Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)



RESEARCH-ARTICLE

PaMpeR: proof method recommendation system for Isabelle/HOL

Authors: Yutaka Nagashima, Yilun He [Authors Info & Affiliations](#)

Publication: ASE 2018: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering • September 2018 • Pages 362–372 • <https://doi.org/10.1145/3238147.3238210>

4 152

Get Access

ABSTRACT

Deciding which sub-tool to use for a given proof state requires expertise specific to each interactive theorem prover (ITP). To mitigate this problem, we present `<pre>PaMpeR</pre>`, a `<U>p</U>roof <U>m</U>ethod <U>r</U>ecommendation system for Isabelle/HOL. Given a proof state, <pre>PaMpeR</pre> recommends proof methods to discharge the proof goal and provides qualitative explanations as to why it suggests these methods. <pre>PaMpeR</pre> generates these recommendations based on existing hand-written proof corpora, thus transferring experienced users' expertise to new users. Our evaluation shows that <pre>PaMpeR</pre> correctly predicts experienced users' proof methods invocation especially when it comes to special purpose proof methods.`

References

1. 2018. PSL with PGT: CICM2018 for Isabelle2017. (2018). <https://github.com/data61/PSL/releases/tag/v0.1.1> To use PaMpeR, one first needs to install Isabelle/HOL, which is distributed at <https://isabelle.in.tum.de/>. PaMpeR: Proof Method Recommendation System for Isabelle/HOL ASE '18, September 3–7, 2018, Montpellier, France

Step1: Proof Method Recommendation (PaMpeR)

The screenshot shows a software interface for formal verification or proof development. The main window displays a text-based proof script in a domain-specific language. The script includes definitions of recursive functions `rev1` and `rev2`, and a theorem stating their equivalence. The word `which_method` is highlighted with a red rectangle, indicating it is the current target for recommendation.

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  | "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  which_method
```

Promising methods for this proof goal are:

simp with expectation of 0.411909082795

auto with expectation of 0.159332679097

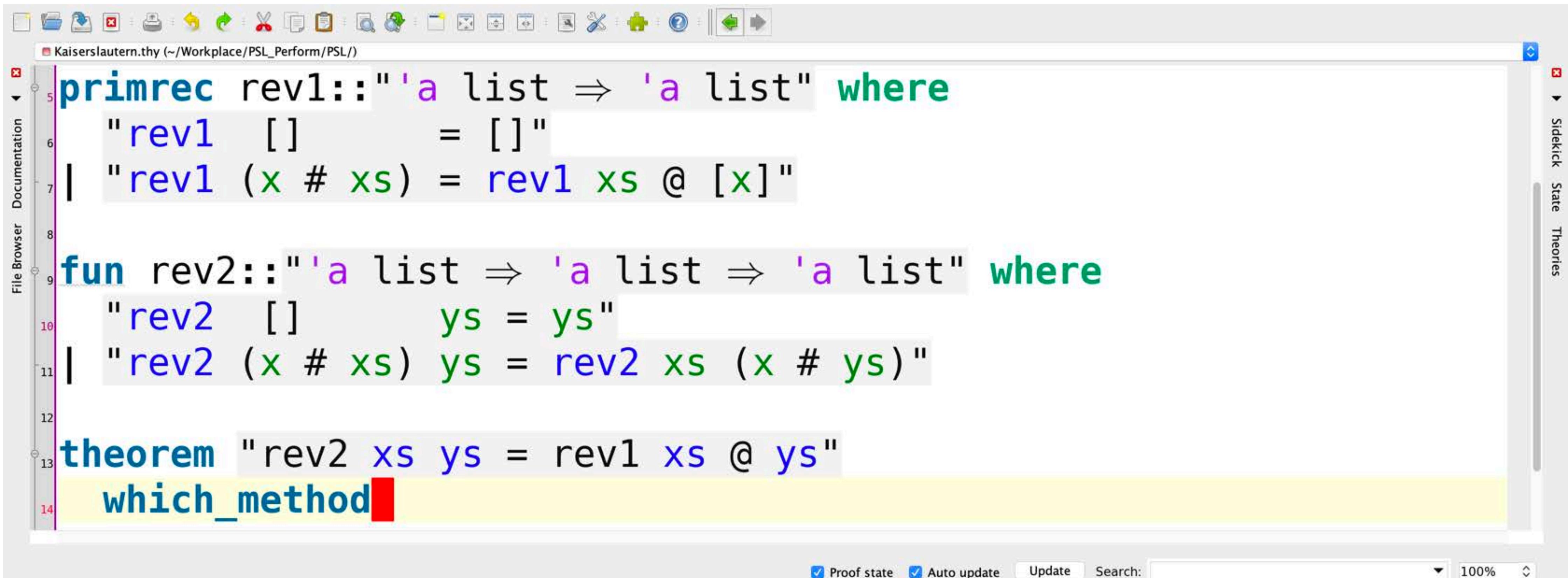
rule with expectation of 0.0874798467373

induction with expectation of 0.0613706657985

metis with expectation of 0.052603838226

induct with expectation of 0.0510162518838

Step1: Proof Method Recommendation (PaMpeR)



```
File Browser Documentation Sidekick State Theories
Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
which_method
```

Proof state Auto update Update Search: 100%

Promising methods for this proof goal are:

simp with expectation of 0.411909082795

auto with expectation of 0.159332679097

rule with expectation of 0.087479846737

induction with expectation of 0.0613706657985

metis with expectation of 0.052603838226

induct with expectation of 0.0510162518838



Arguments of tactics?

Path to automatic induction



Path to automatic induction

Step 1: Tactic recommendation.



Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.



Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).

b: Logical Feature Extraction (LiFtEr).

c: smart_induct using LiFtEr.

Step 3: Putting things together.



```
fun rev2 :: "'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []          ys = ys"
  "rev2 (x # xs)   ys = rev2 xs (x # ys)"
```

Proof state Auto update Search: 100%

consts
 rev2 :: "'a list ⇒ 'a list ⇒ 'a list"
Found termination order: " $(\lambda p. \text{length} (\text{fst } p)) <^{*\text{mlex}*} \{\}$ "

The screenshot shows the Isabelle proof assistant interface. The top part displays the theory file `Kaiserslautern.thy` with the following content:

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []          ys = ys"
  "rev2 (x # xs)   ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
```

The line `theorem "rev2 xs ys = rev1 xs @ ys"` is highlighted with a yellow background. The bottom part of the interface shows the proof state:

```
proof (prove)
goal (1 subgoal):
  1. rev2 xs ys = rev1 xs @ ys
```

Proof state Auto update Search: 100%

```
proof (prove)
goal (1 subgoal):
  1. rev2 xs ys = rev1 xs @ ys
```

The screenshot shows the Isabelle/Isar interface with the file `Kaiserslautern.thy` open. The code defines a function `rev2` and a theorem about it.

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []          ys = ys"
  "rev2 (x # xs)   ys = rev2 xs (x # ys)"

strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved](*PSL*)

theorem "rev2 xs ys = rev1 xs @ ys"
```

The code is color-coded: `fun`, `where`, and `strategy` are blue; `rev2`, `xs`, `ys`, and `x` are green; and `]`, `(*PSL*)`, and `theorem` are black. The `strategy` line is highlighted with a yellow background. The `theorem` line is also highlighted with a yellow background. The interface includes a toolbar at the top, a vertical file browser on the left, and a vertical status bar on the right. At the bottom, there are tabs for Output, Query, Sledgehammer, and Symbols, along with a status bar showing file statistics and memory usage.

The screenshot shows the Isabelle/Isar interface with the file `Kaiserslautern.thy` open. The code defines a function `rev2` and a theorem `rev2_xs_ys`. The `find_proof` command is being typed at the end of the theorem statement.

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []          ys = ys"
  "rev2 (x # xs)   ys = rev2 xs (x # ys)"

strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved] (*PSL*)

theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd
oops
```

The interface includes a toolbar at the top, a vertical file browser on the left, and a sidebar on the right. The status bar at the bottom shows file statistics and a timestamp.

The screenshot shows the Isabelle IDE interface with a theory file named `Kaiserslautern.thy`. The code defines a function `rev2` and a theorem `rev2_xs_ys`.

```
File Browser Documentation
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []          ys = ys"
  "rev2 (x # xs)   ys = rev2 xs (x # ys)"

strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved] (*PSL*)

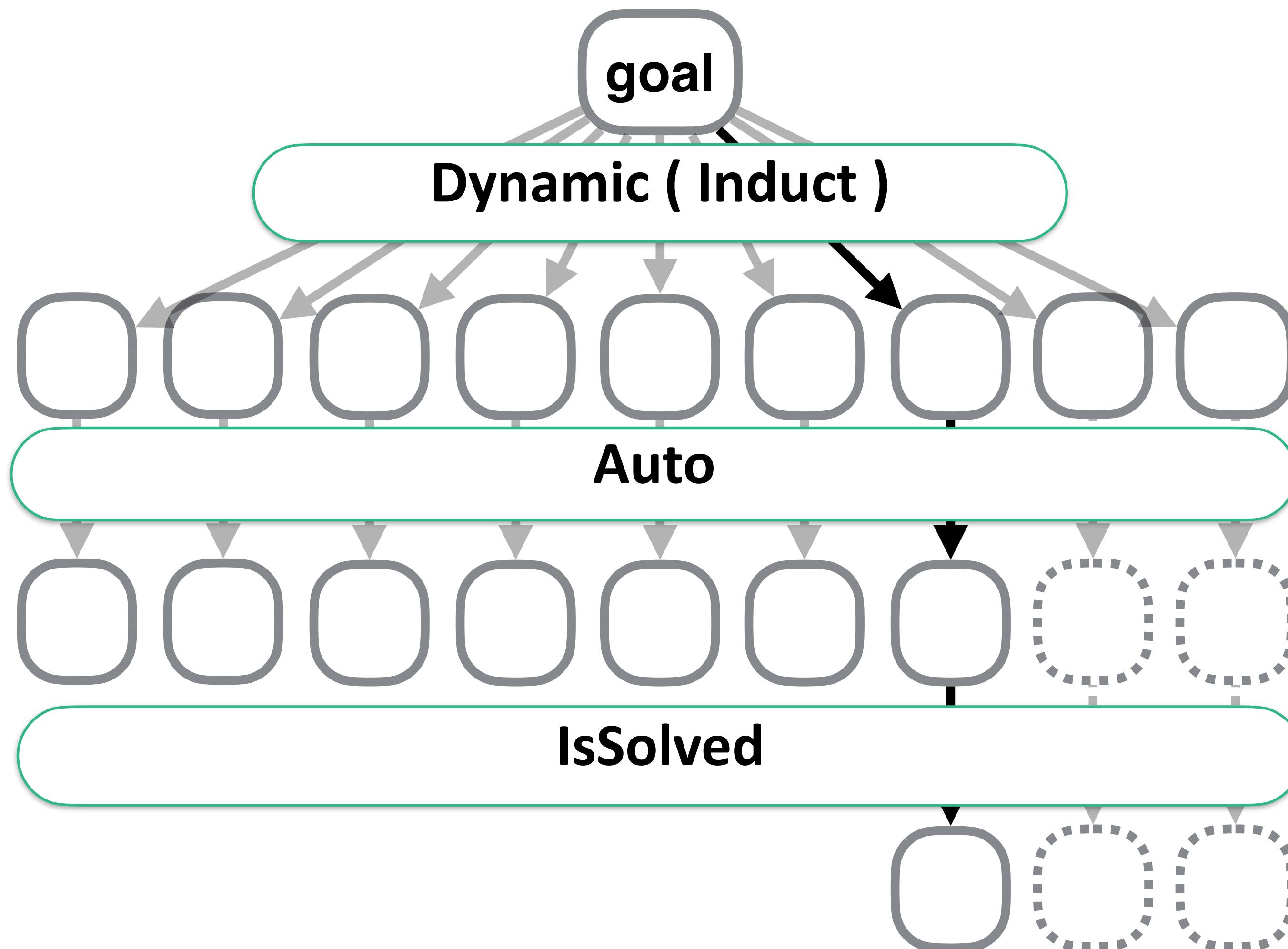
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd
oops
```

Proof state Auto update Update Search: 100%

```
apply (induct xs arbitrary: ys)
apply auto
done
```

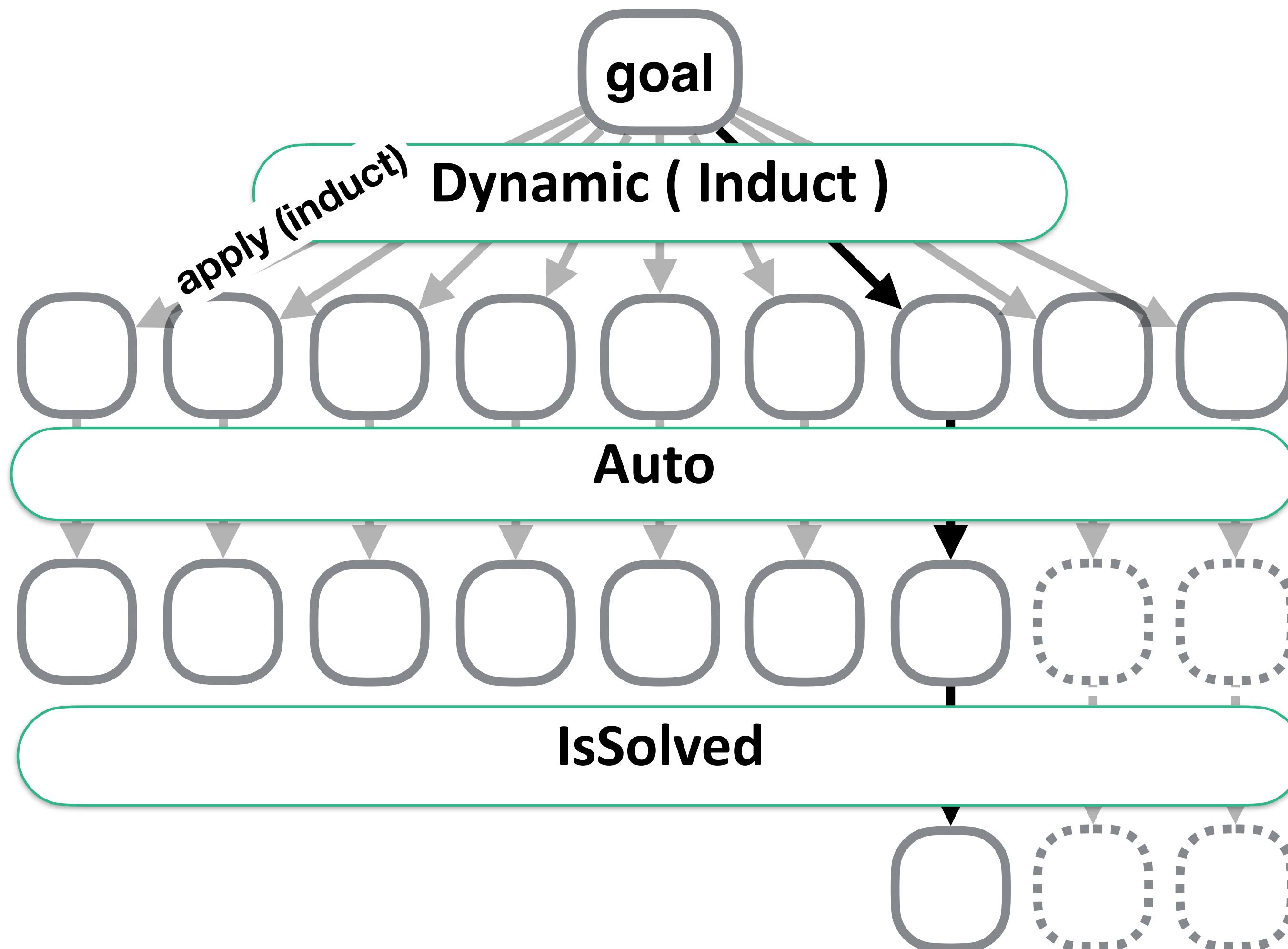
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



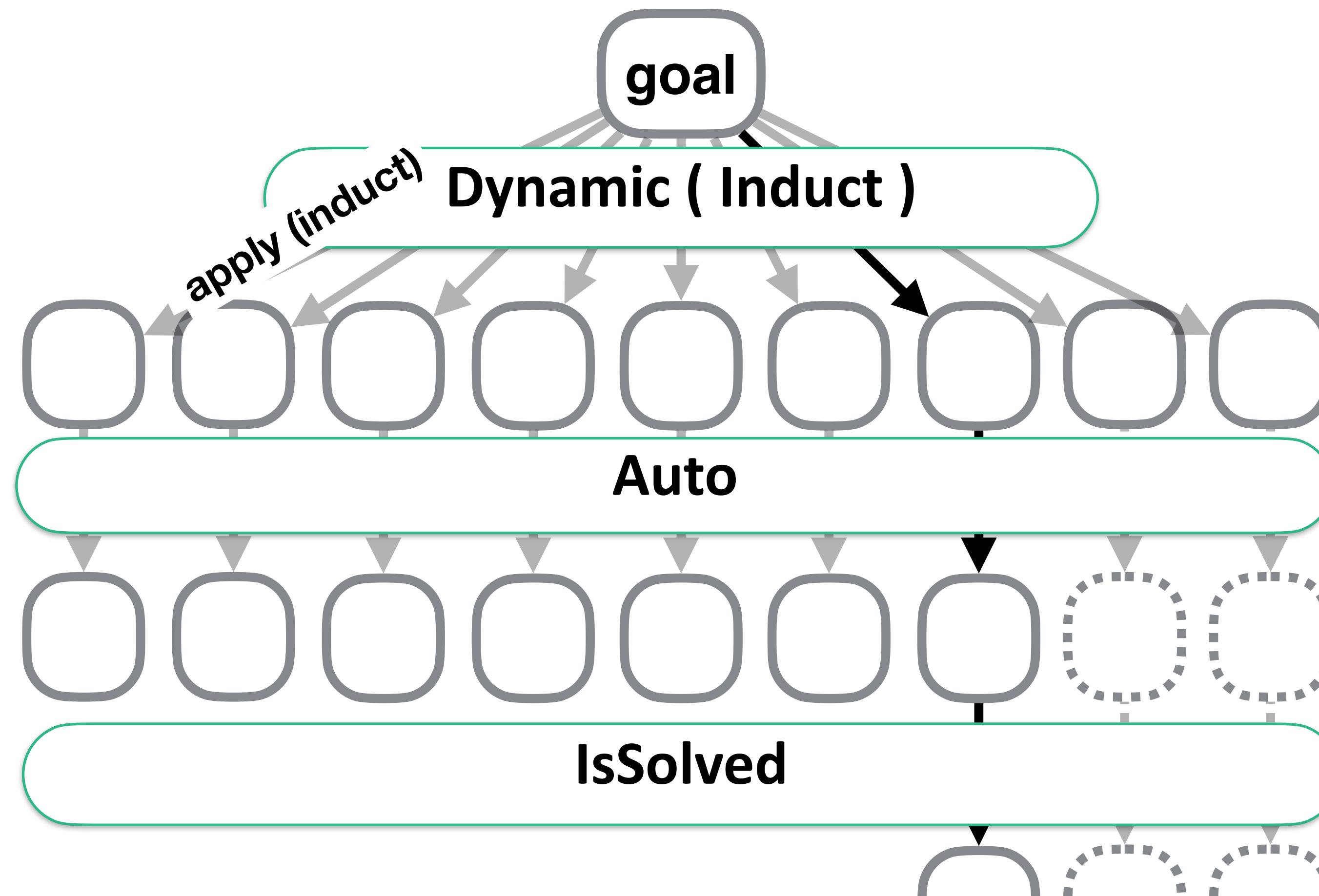
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```

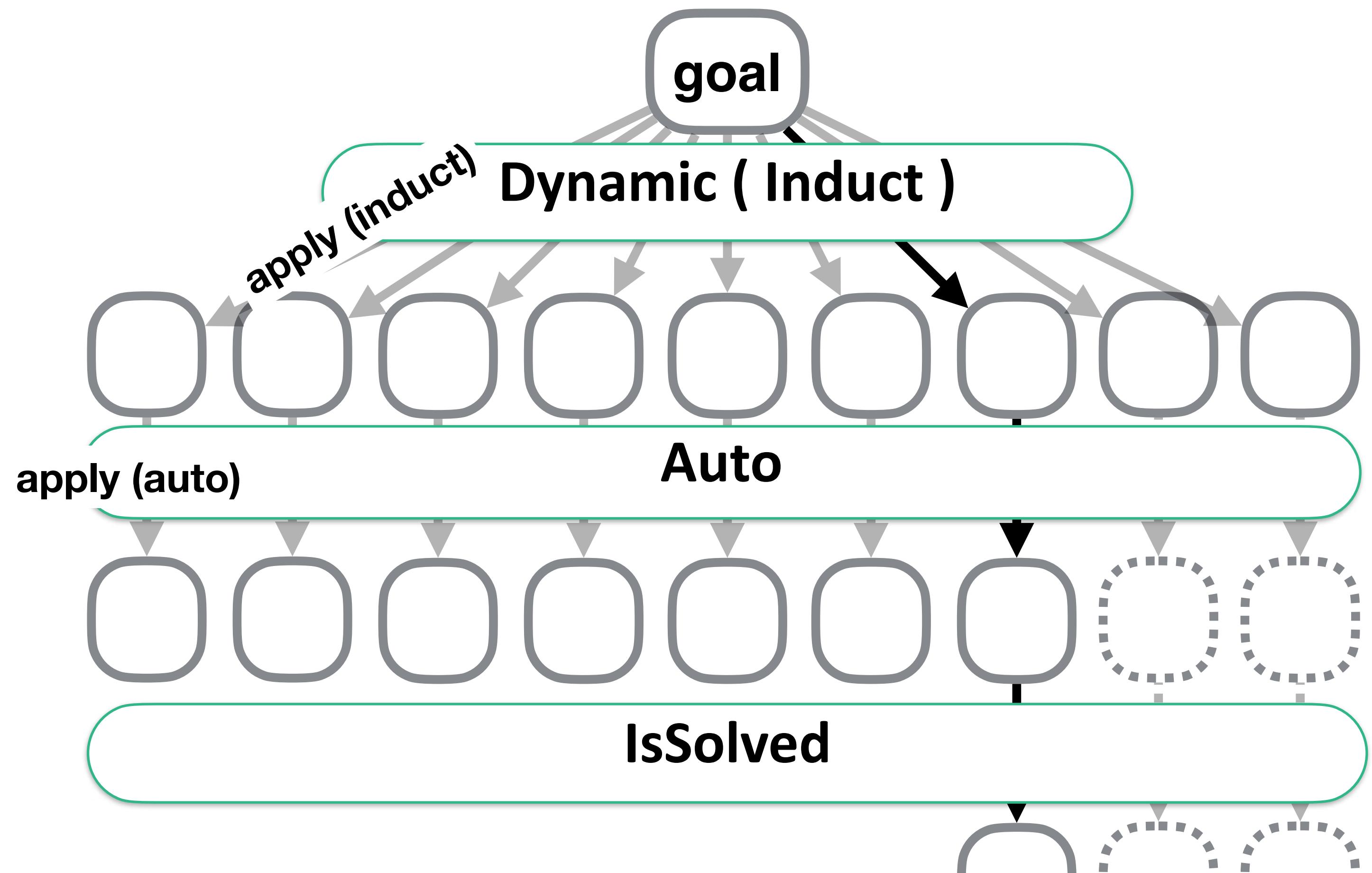


```
goal (1 subgoal):
```

```
1.  $\wedge y. y \in \{F. \text{is\_filter } F\} \Rightarrow \text{rev2 } xs \text{ } ys = \text{rev1 } xs @ ys$ 
```

```
theorem "rev2 xs ys = rev1 xs @ ys"
```

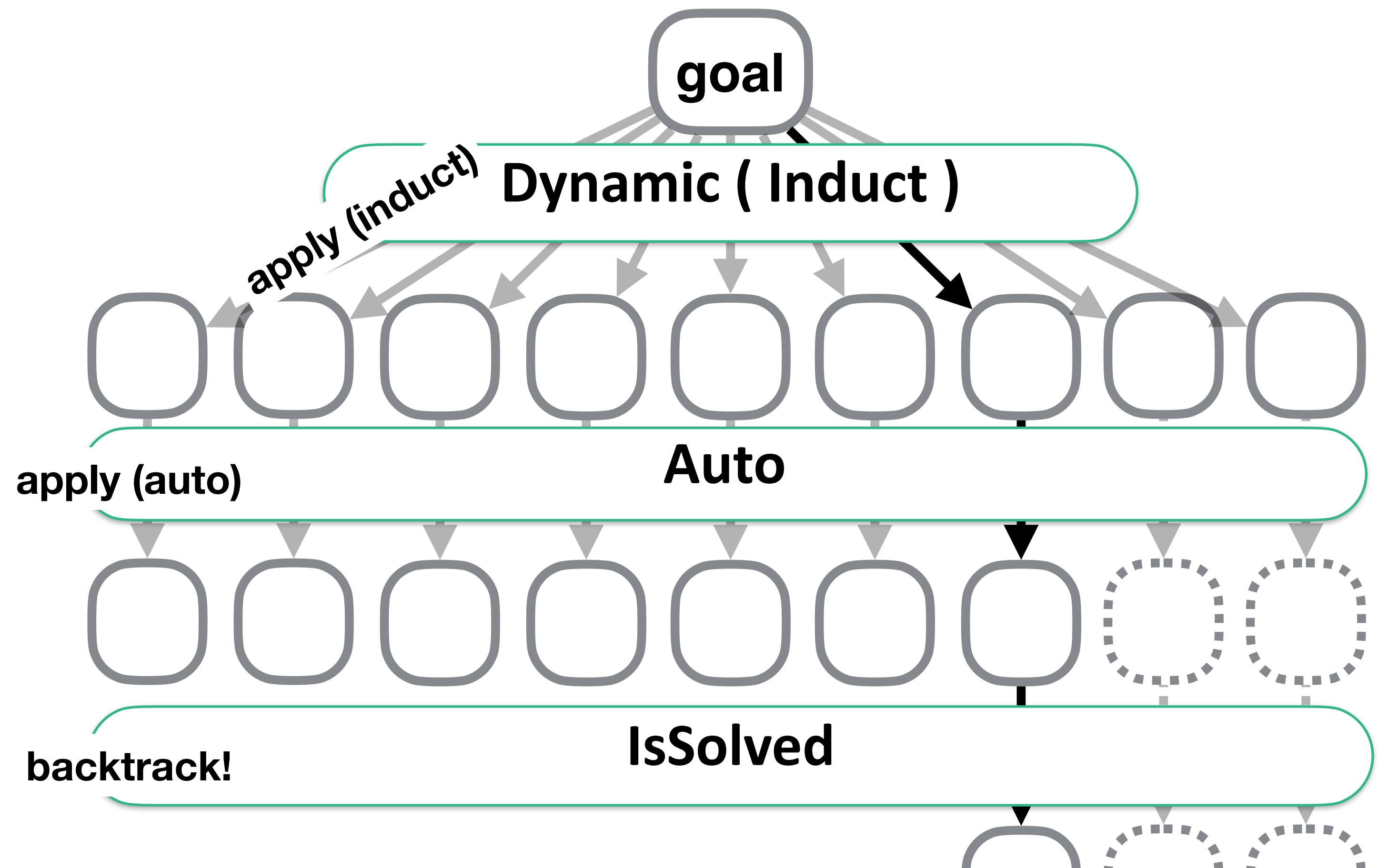
```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
goal (1 subgoal):  
1.  $\wedge y. \text{is\_filter } y \Rightarrow \text{rev2 } xs \text{ } ys = \text{rev1 } xs @ ys$ 
```

```
theorem "rev2 xs ys = rev1 xs @ ys"
```

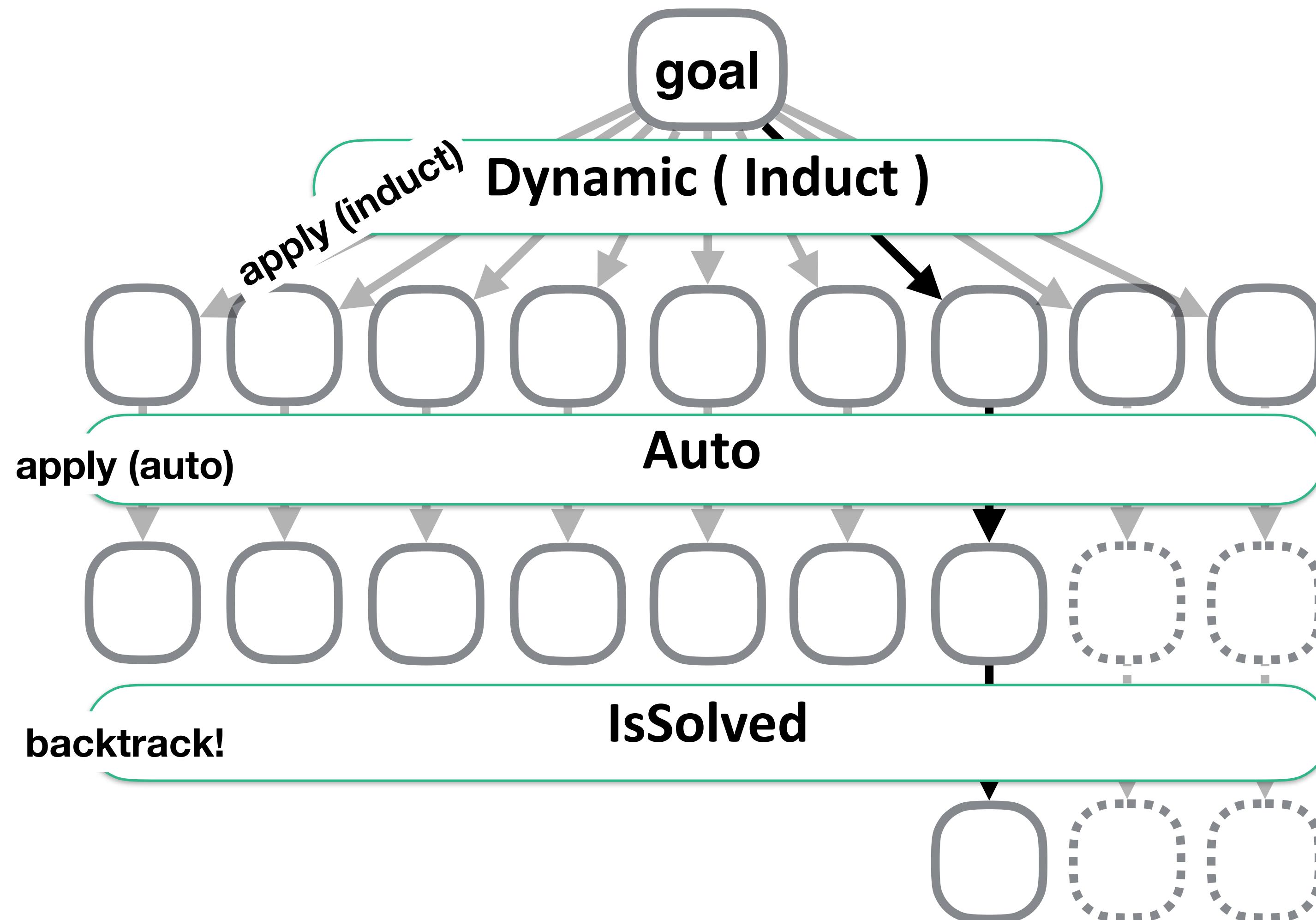
```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
goal (1 subgoal):  
1.  $\wedge y. \text{is\_filter } y \Rightarrow \text{rev2 } xs \text{ } ys = \text{rev1 } xs @ ys$ 
```

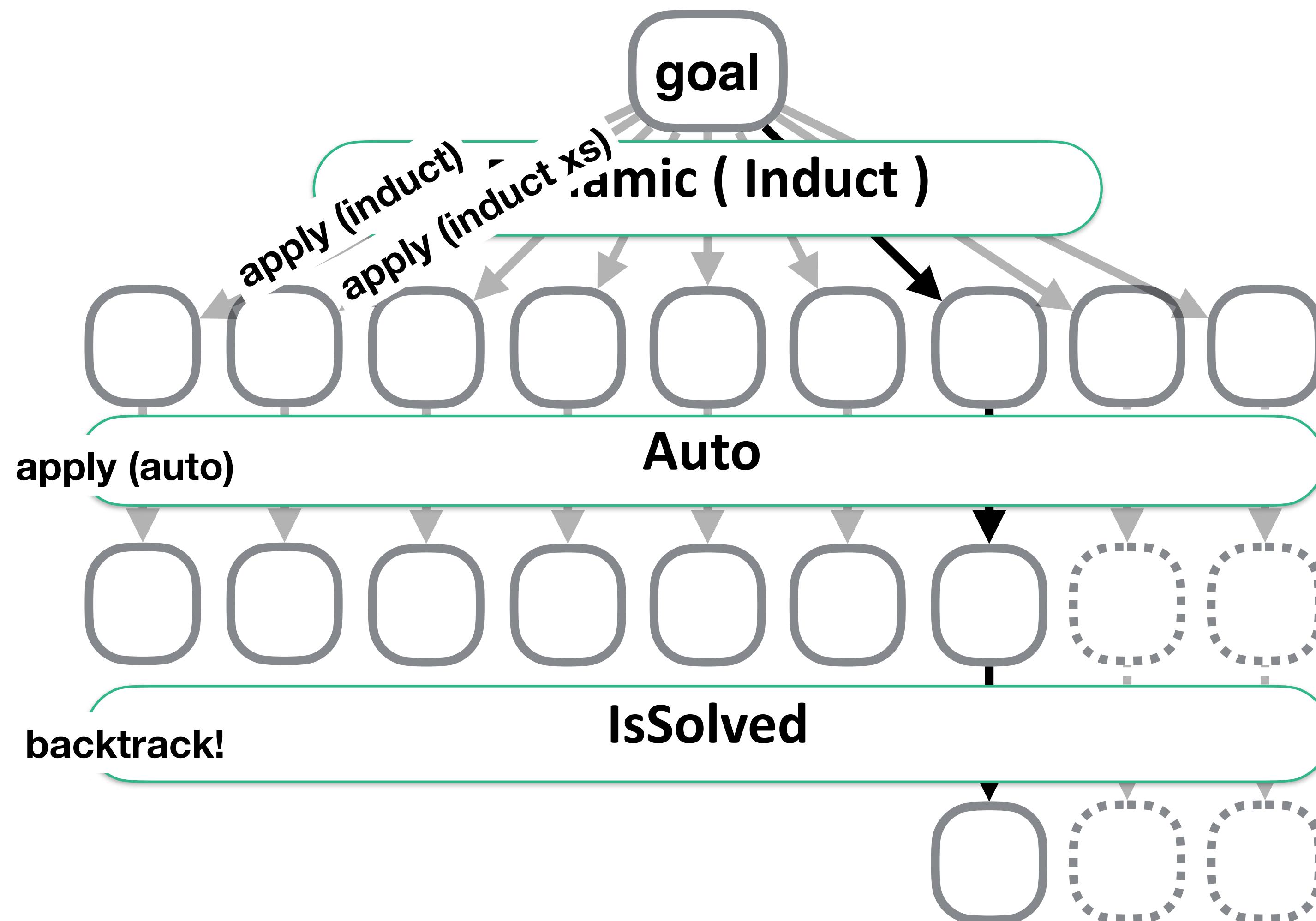
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



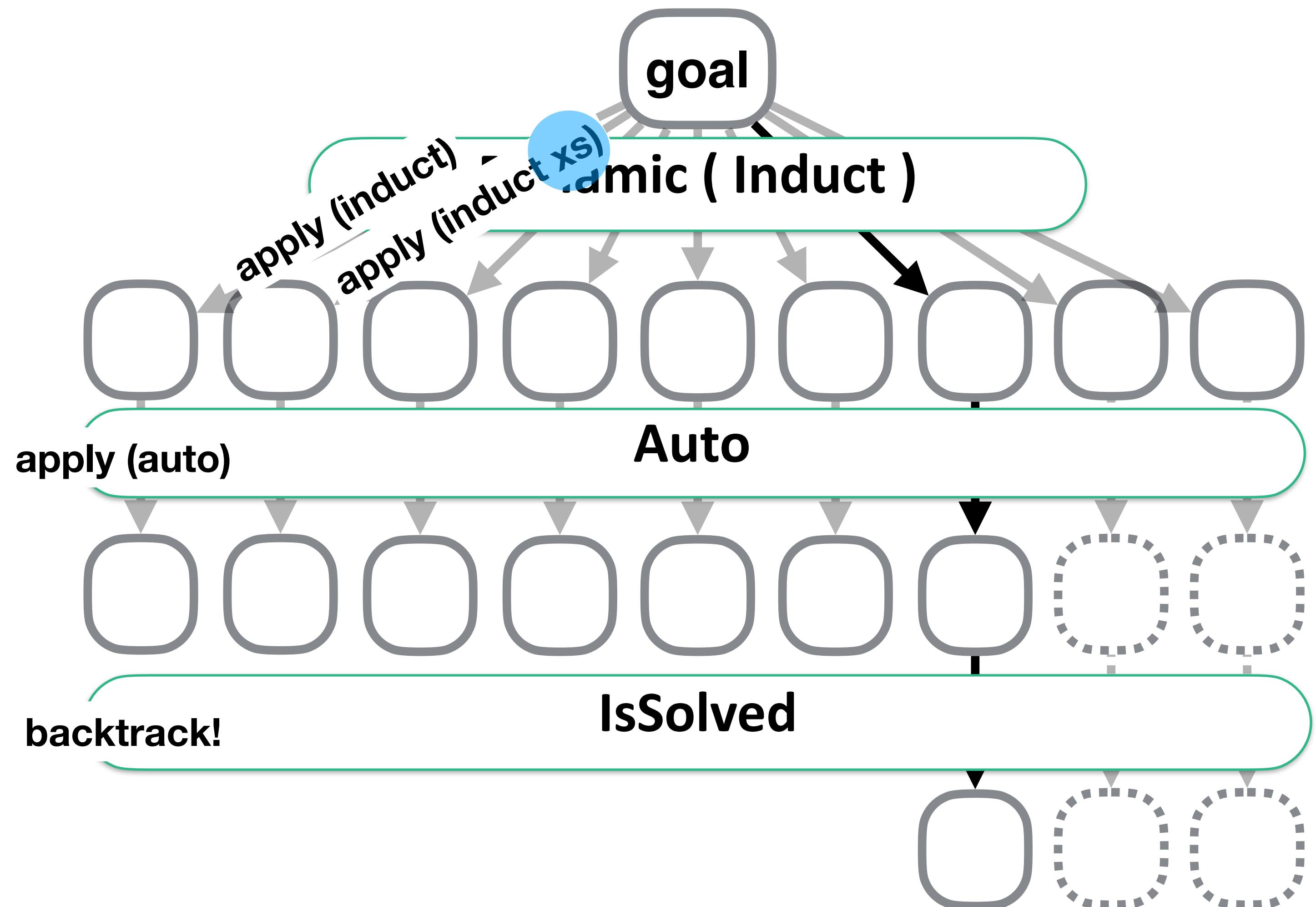
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



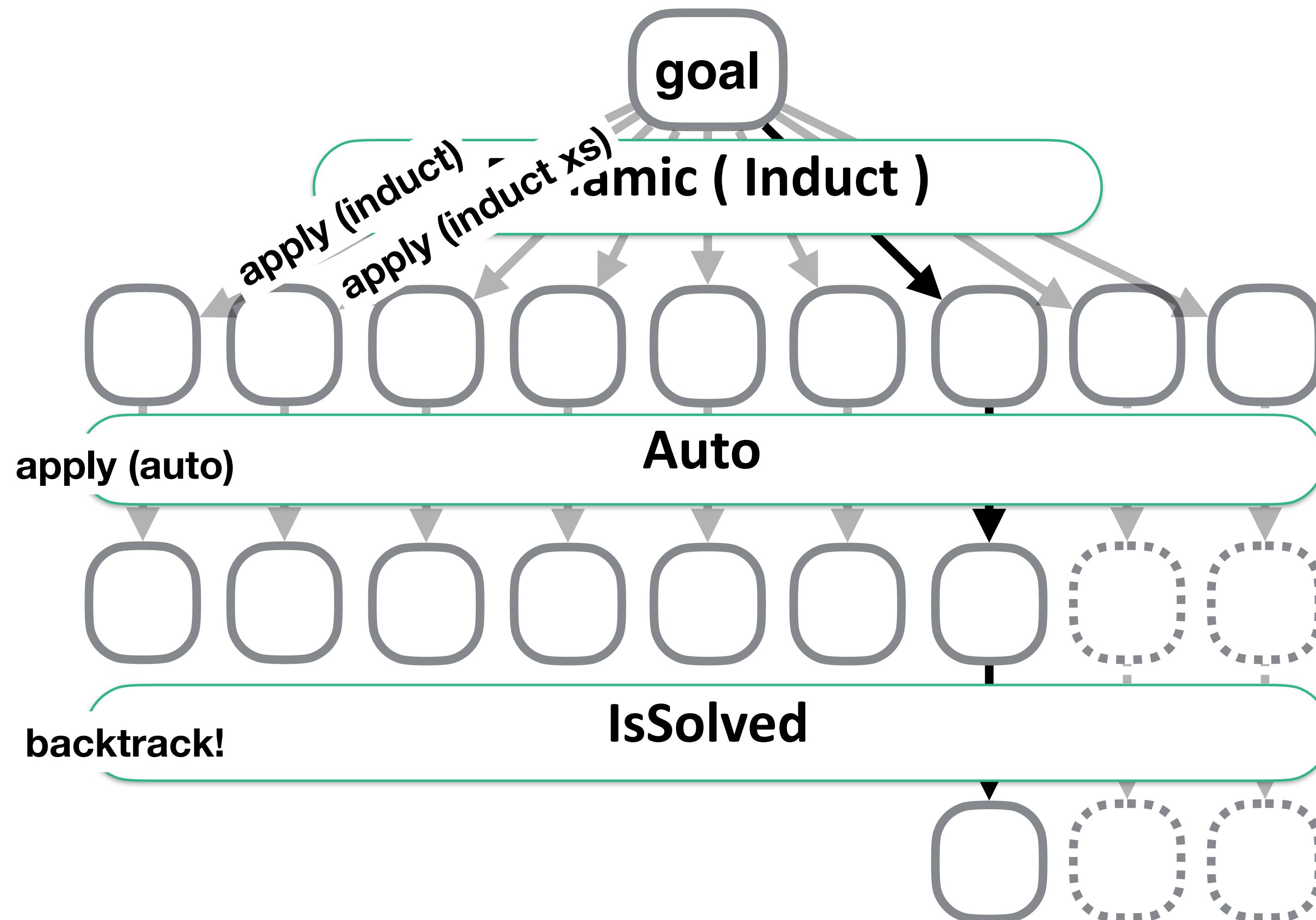
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



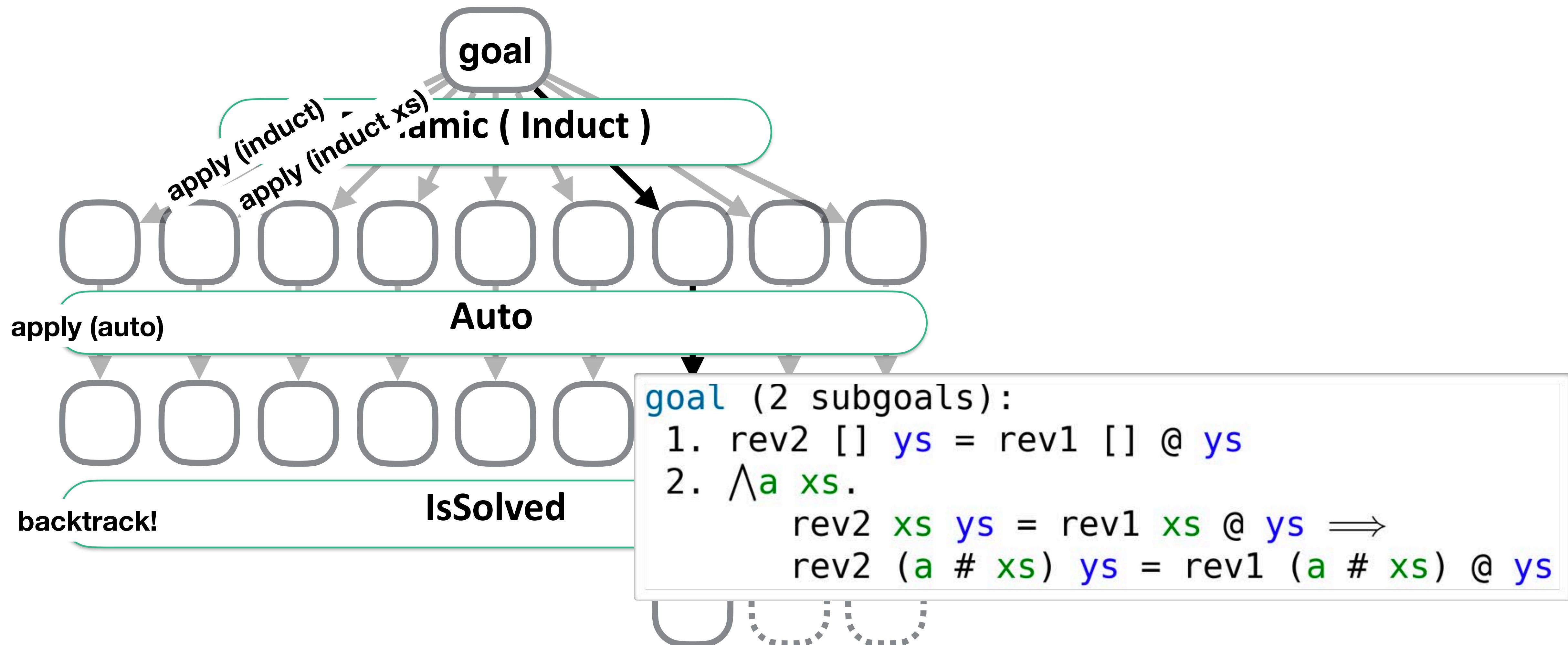
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



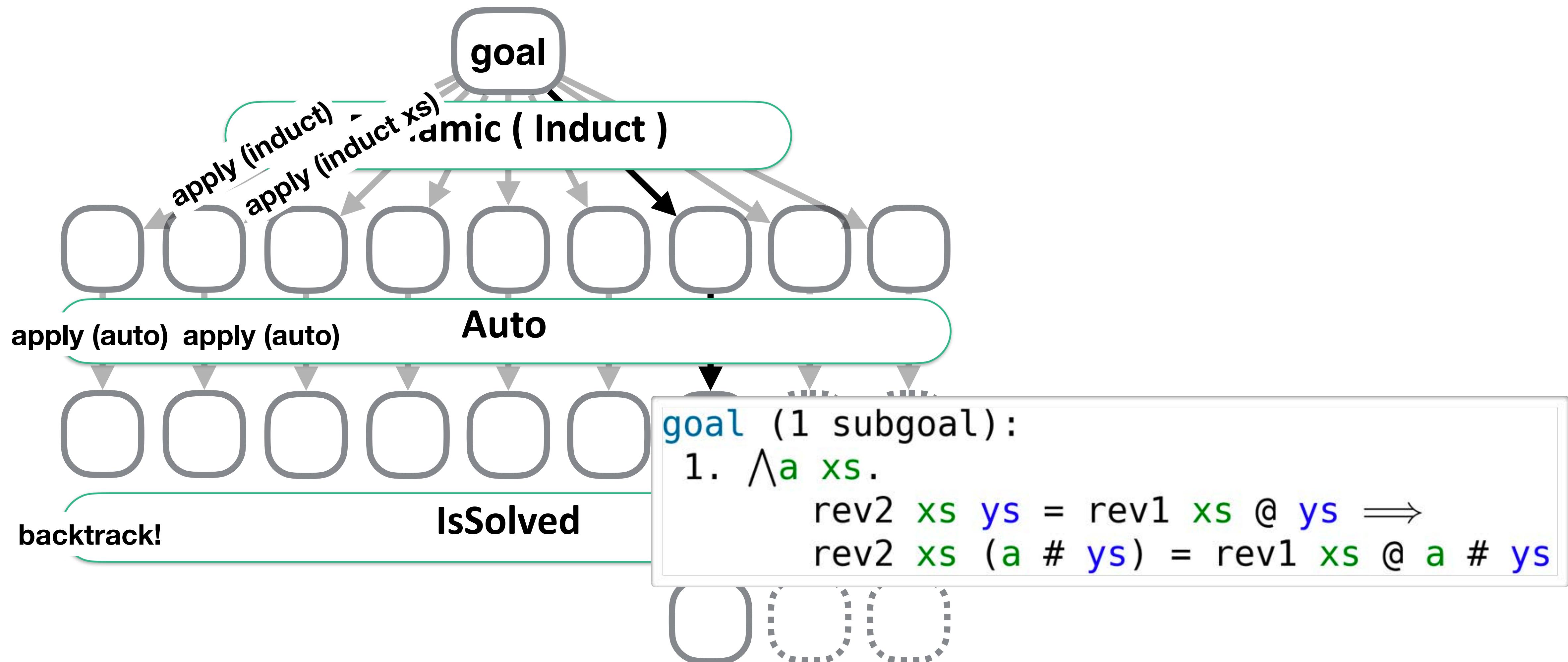
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



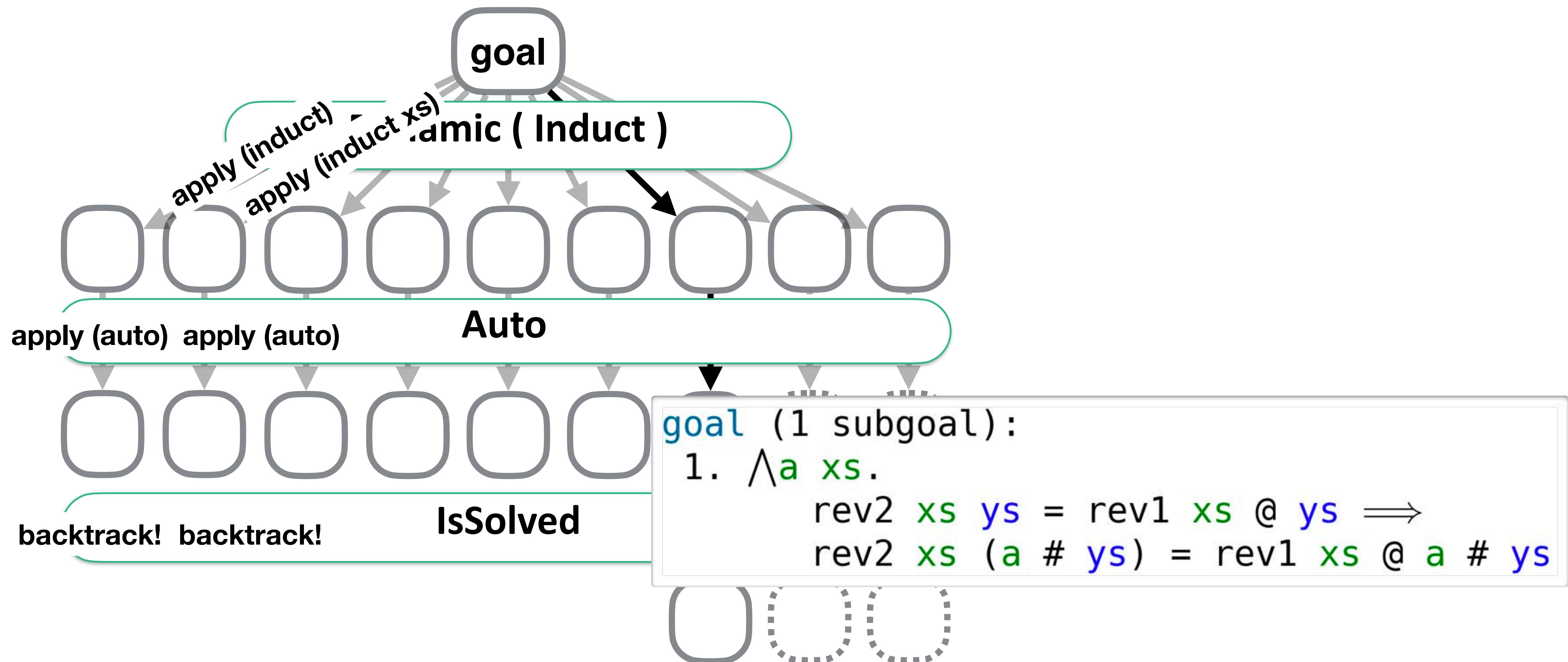
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



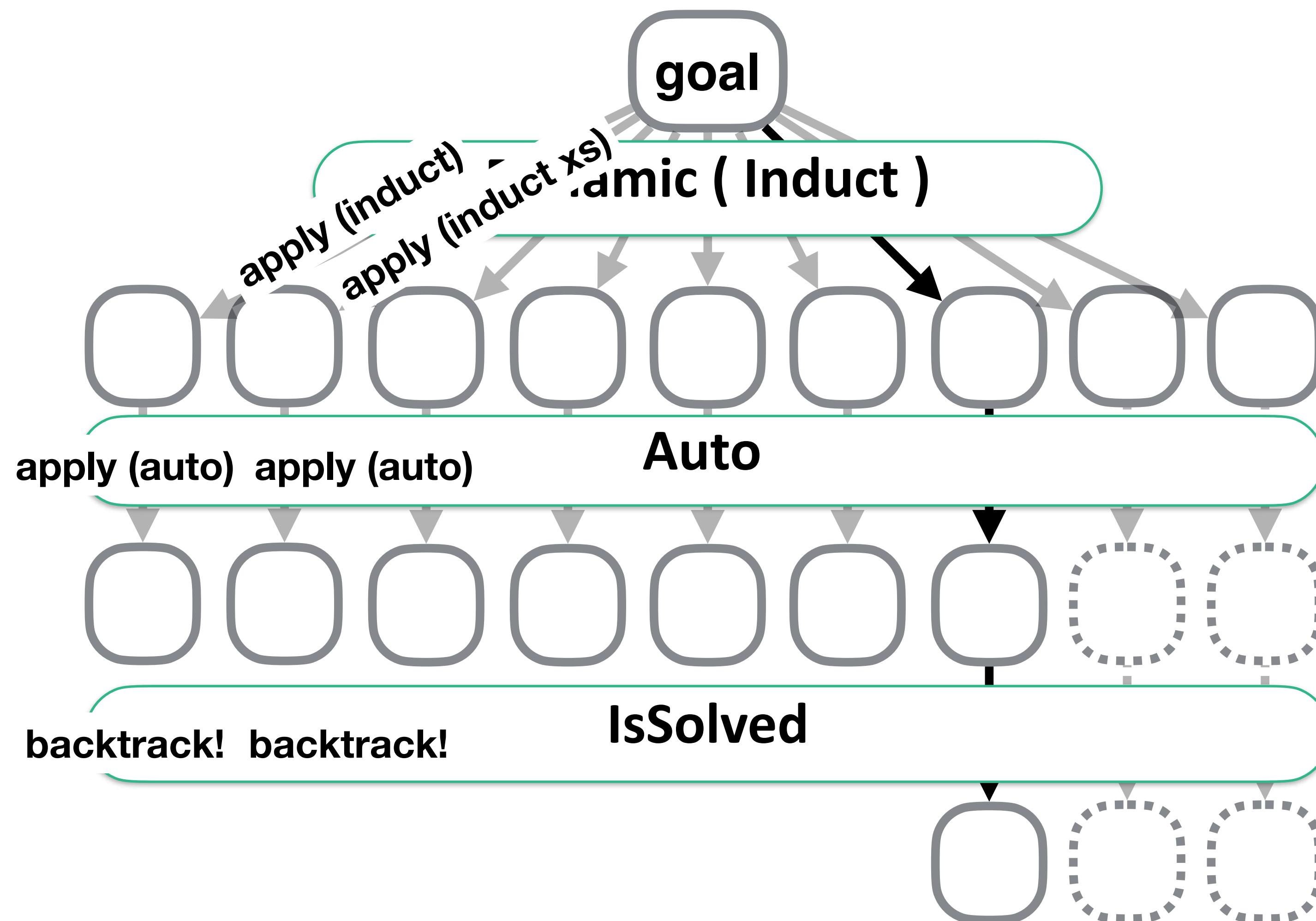
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



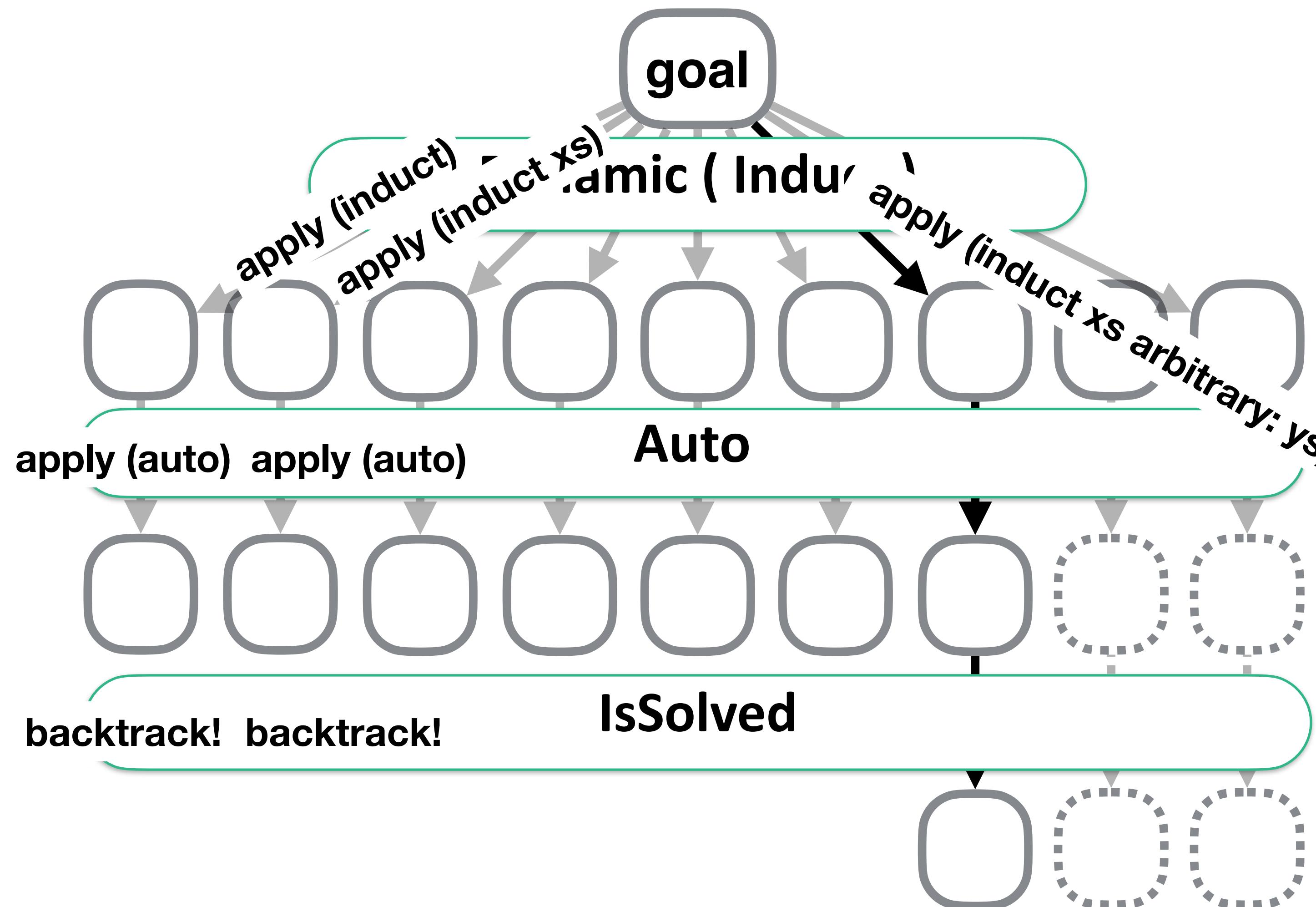
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



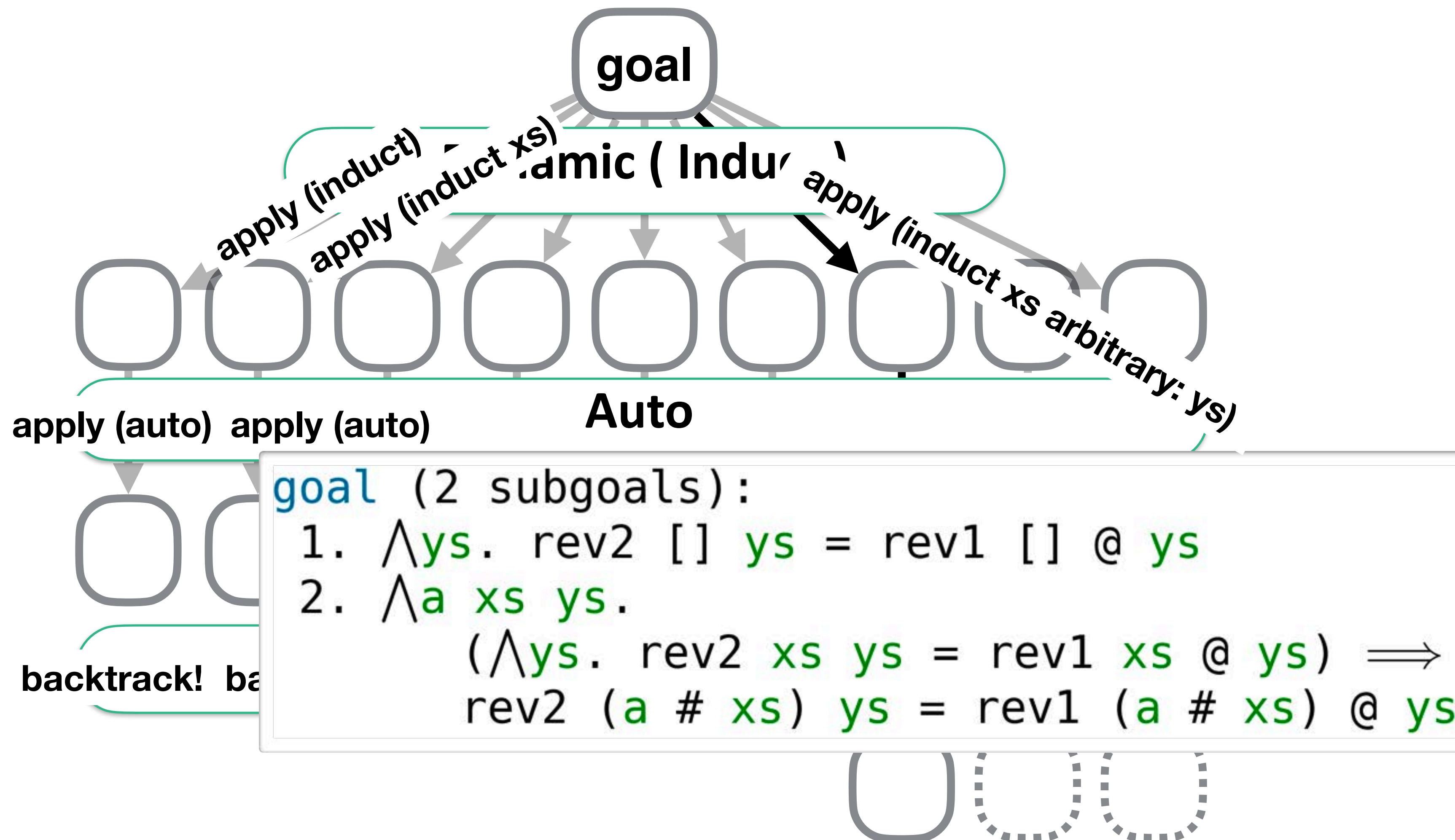
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



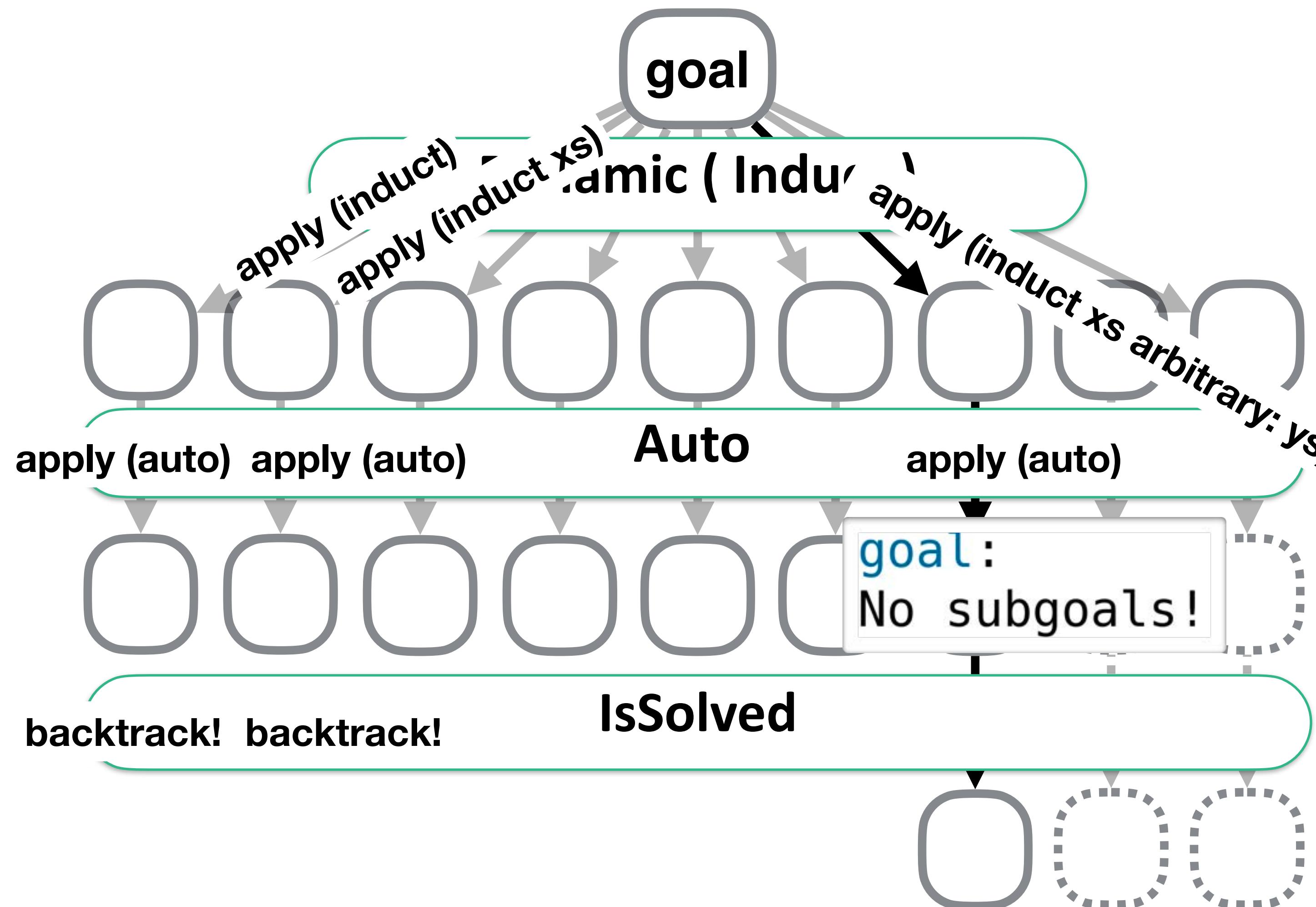
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



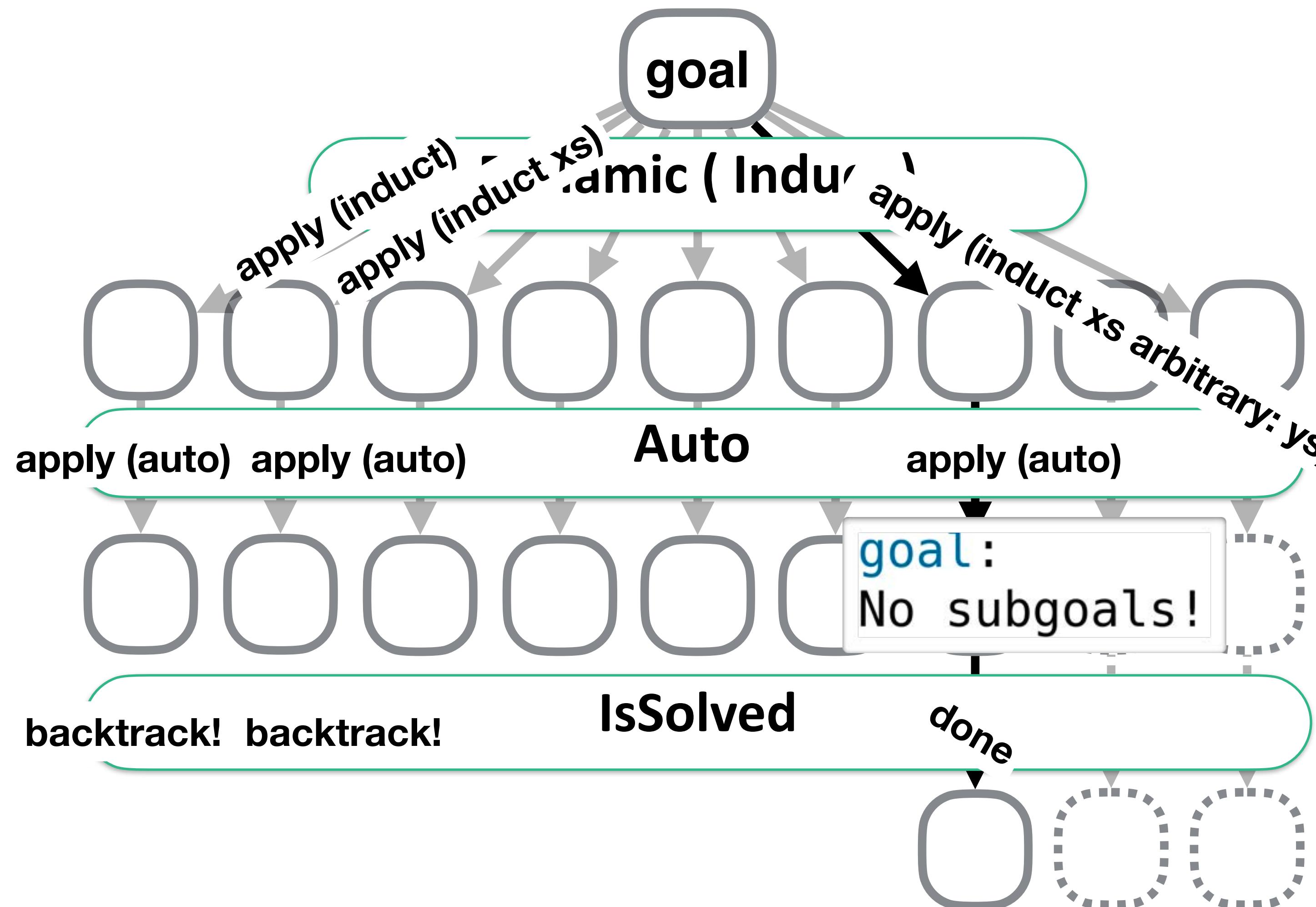
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



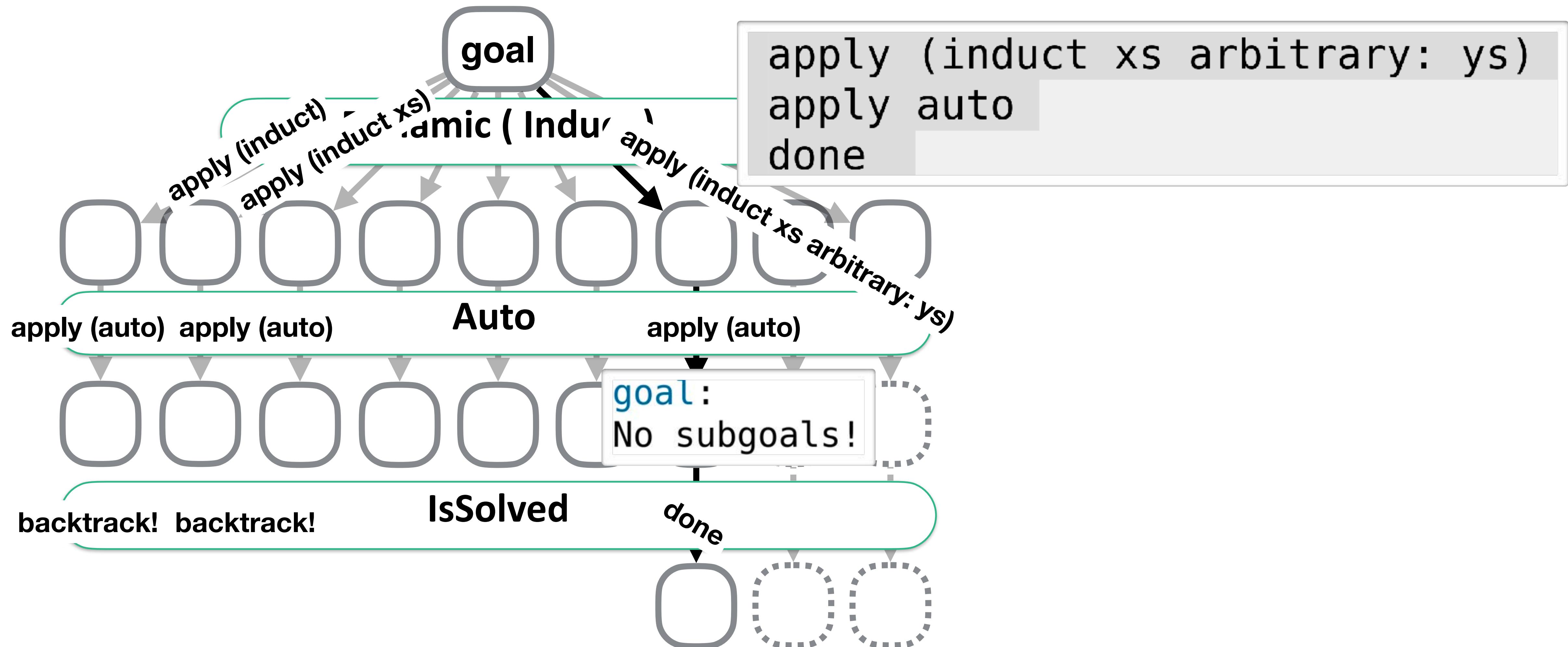
```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
```

```
find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
File Browser Documentation
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []          ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved] (*PSL*)

theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd
oops
```

Proof state Auto update Update Search: 100%

```
apply (induct xs arbitrary: ys)
apply auto
done
```



Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
15 theorem "rev2 xs ys = rev1 xs @ ys"  
16 find_proof DInd  
17 oops
```



```
apply (induct xs arbitrary: ys)  
apply auto  
done
```

Proof state Auto update Update Search:

Output Query Sledgehammer Symbols

16,18 (381/393)

(isabelle,isabelle,UTF-8-Isabelle)| nmr o U.. 195/512MB 1 error(s) 2:14 PM



Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
15 theorem "rev2 xs ys = rev1 xs @ ys"  
16 find_proof DInd  
17 oops
```



```
apply (induct xs arbitrary: ys)  
apply auto  
done
```

Proof state Auto update Update Search:

Output Query Sledgehammer Symbols

16,18 (381/393)

(isabelle,isabelle,UTF-8-Isabelle)| nmr o U.. 195/512MB 1 error(s) 2:14 PM



Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)



```
15 theorem "rev2 xs ys = rev1 xs @ ys"
```

?find proof DInd
oops

Indeed, auto returned quickly in our case.

```
| apply (induct xs arbitrary: ys)  
apply auto  
done
```



Proof state Auto update Update Search:



Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)



```
15 theorem "rev2 xs ys = rev1 xs @ ys"
```

```
?find proof DInd  
oops
```

```
apply (apply auto done
```

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?





Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them **quickly**.

(*PSL*)



```
15 theorem "rev2 xs ys = rev1 xs @ ys"
```

```
?find proof DInd  
oops
```

```
apply (apply auto  
done
```

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

We have to identify promising arguments for induction tactics

without relying on a search!





Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them **quickly**.

(*PSL*)



theorem "rev2 xs ys = rev1 xs @ ys"

?find proof DInd
oops

apply (apply auto
done

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

We have to identify promising arguments for induction tactics

without relying on a search!

Deep Learning?



Many key challenges remain

Unsupervised Learning

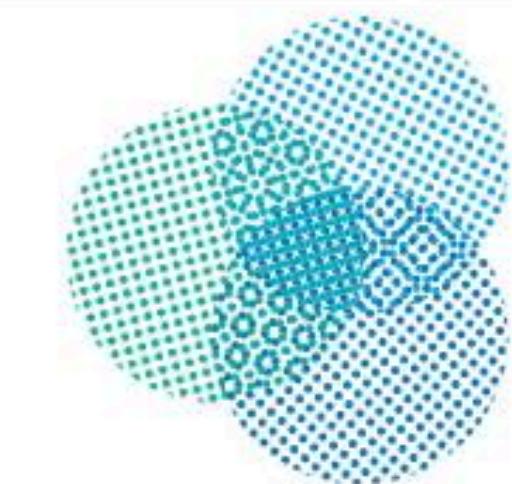
Memory and one-shot learning

Imagination-based Planning with
Generative Models

Learning Abstract Concepts

Transfer Learning

Language understanding



CENTER FOR
Brains
Minds +
Machines

March 20, 2019

The Power of
Self-Learning Systems

Demis Hassabis
DeepMind

Many key challenges remain

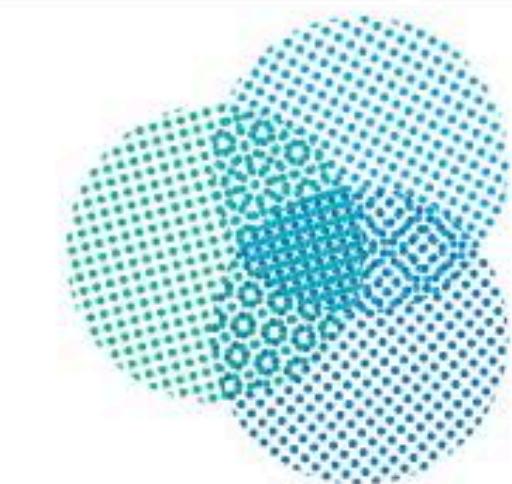
Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with
Generative Models

Learning Abstract Concepts

Abstract concepts?



CENTER FOR
Brains
Minds +
Machines

March 20, 2019

The Power of
Self-Learning Systems

Demo

Deepw



Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with
Generative Models

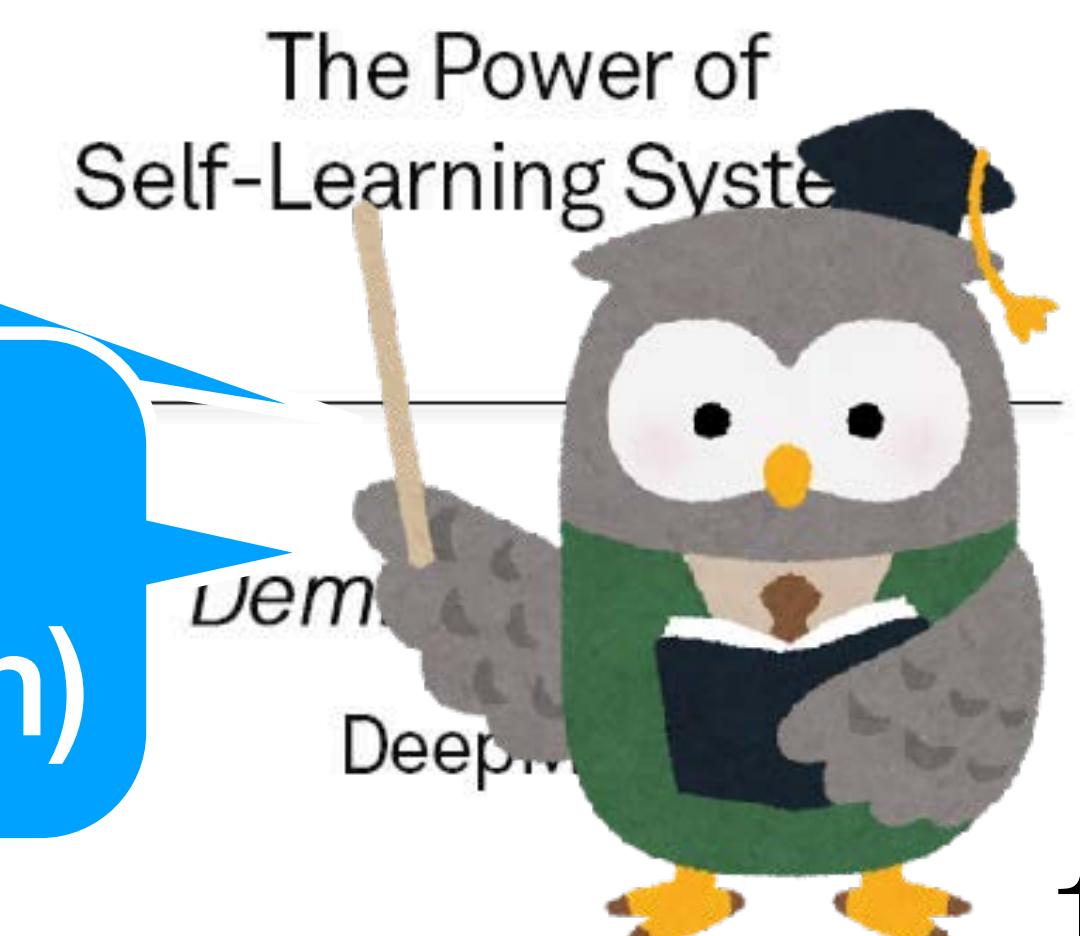
Learning Abstract Concepts

Abstract concepts?

Why not logic?
(LiFtEr = Logical Feature Extraction)



March 20, 2019



Abstract Syntax of LiFtEr

```
assertion := atomic | connective | quantifier | ( assertion )
type := term | term_occurrence | rule | number
modifier_term := induction_term | arbitrary_term
quantifier :=  $\exists x : type.$  assertion
            |  $\forall x : type.$  assertion
            |  $\exists x : term \in modifier\_term .$  assertion
            |  $\forall x : term \in modifier\_term .$  assertion
            |  $\exists x : rule .$  assertion
            |  $\exists x : term\_occurrence \in y : term .$  assertion
            |  $\forall x : term\_occurrence \in y : term .$  assertion
connective := True | False | assertion  $\vee$  assertion | assertion  $\wedge$  assertion
              | assertion  $\rightarrow$  assertion |  $\neg$  assertion
pattern := all_only_var | all_constructor | mixed
atomic :=
rule is_rule_of term_occurrence
| term_occurrence term_occurrence_is_of_term term
| are_same_term ( term_occurrence , term_occurrence )
| term_occurrence is_in_term_occurrence term_occurrence
| is_atomic term_occurrence
| is_constant term_occurrence
| is_recursive_constant term_occurrence
| is_variable term_occurrence
| is_free_variable term_occurrence
| is_bound_variable term_occurrence
| is_lambda term_occurrence
| is_application term_occurrence
| term_occurrence is_an_argument_of term_occurrence
| term_occurrence is_nth_argument_of term_occurrence
```

Abstract Syntax of LiFtEr

```
assertion := atomic | connective | quantifier | ( assertion )
type := term | term_occurrence | rule | number
modifier_term := induction_term | arbitrary_term
quantifier :=  $\exists x : type.$  assertion
            |  $\forall x : type.$  assertion
            |  $\exists x : term \in modifier\_term .$  assertion
            |  $\forall x : term \in modifier\_term .$  assertion
            |  $\exists x : rule .$  assertion
            |  $\exists x : term\_occurrence \in y : term .$  assertion
            |  $\forall x : term\_occurrence \in y : term .$  assertion
connective := True | False | assertion  $\vee$  assertion
              | assertion  $\wedge$  assertion | assertion  $\rightarrow$  assertion |  $\neg$  assertion
pattern := all_only_var | all_constructor
atomic :=
    rule_is_rule_of_term_occurrence
    | term_occurrence_term_occurrences
    | are_same_term ( term_occurrence , term_occurrence )
    | term_occurrence_is_in_term_occurrence term_occurrence
    | is_atomic term_occurrence
    | is_constant term_occurrence
    | is_recursive_constant term_occurrence
    | is_variable term_occurrence
    | is_free_variable term_occurrence
    | is_bound_variable term_occurrence
    | is_lambda term_occurrence
    | is_application term_occurrence
    | term_occurrence_is_an_argument_of term_occurrence
    | term_occurrence_is_nth_argument_of term_occurrence
```

Example Assertion in LiFtEr (in Abstract Syntax)

```
∃ r1 : rule. True
→
∃ r1 : rule.
  ∃ t1 : term.
    ∃ to1 : term_occurrence ∈ t1 : term.
      r1 is_rule_of to1
    ∧
      ∀ t2 : term ∈ induction_term.
        ∃ to2 : term_occurrence ∈ t2 : term.
          ∃ n : number.
            is_nth_argument_of (to2, n, to1)
          ∧
            t2 is_nth_induction_term n
```

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$



$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$



$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$



conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$



$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas

$\exists r1 : \text{rule}.$ ←

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge ← conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ ← variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge ← conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ ← variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$ ← variable for term occurrences

\wedge ← conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ ← variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$ ← variable for term occurrences

\wedge ← conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$ ← variable for natural numbers

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ ← variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$ ← variable for term occurrences

\wedge ← conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$ ← variable for natural numbers

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication existential quantifier

$\exists r1 : \text{rule}. \text{True} \rightarrow \exists r1 : \text{rule}. \exists t1 : \text{term}. \exists to1 : \text{term_occurrence} \in t1 : \text{term}. r1 \text{ is_rule_of } to1 \wedge \forall t2 : \text{term} \in \text{induction_term}. \exists to2 : \text{term_occurrence} \in t2 : \text{term}. \exists n : \text{number}. \text{is_nth_argument_of} (to2, n, to1) \wedge t2 \text{ is_nth_induction_term } n$

variable for auxiliary lemmas

variable for terms

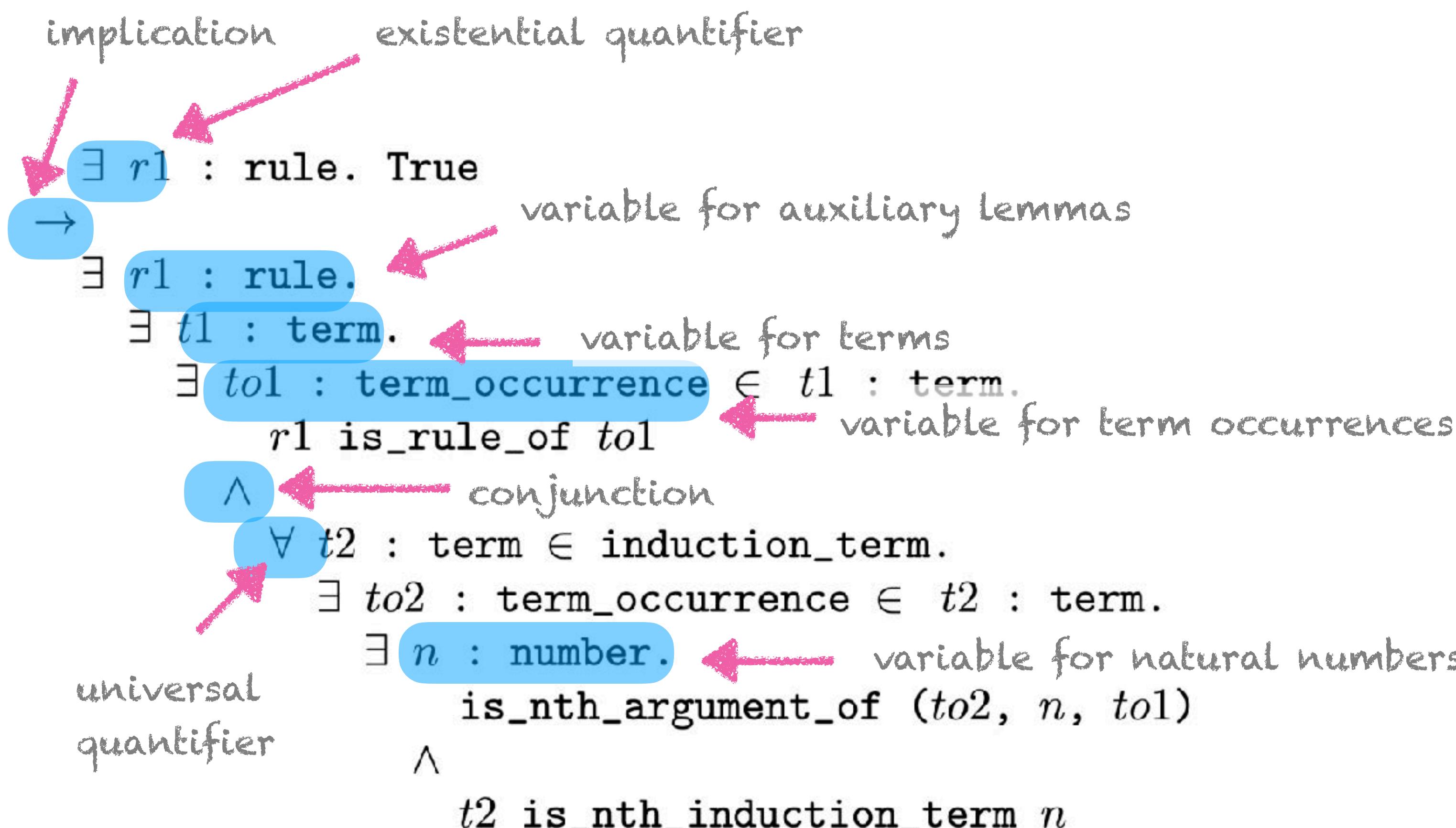
variable for term occurrences

conjunction

universal quantifier

Example Assertion in LiFtEr (in Abstract Syntax)

LiFtEr assertion: (proof goal * induction arguments) -> bool



```

primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  | "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
  apply auto done

 $\exists r1 : \text{rule}. \text{True}$ 
 $\rightarrow$ 
 $\exists r1 : \text{rule}.$ 
 $\exists t1 : \text{term}.$ 
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$ 
 $r1 \text{ is\_rule\_of } to1$ 
 $\wedge$ 
 $\forall t2 : \text{term} \in \text{induction\_term}.$ 
 $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$ 
 $\exists n : \text{number}.$ 
 $\text{is\_nth\_argument\_of } (to2, n, to1)$ 
 $\wedge$ 
 $t2 \text{ is\_nth\_induction\_term } n$ 

```

```

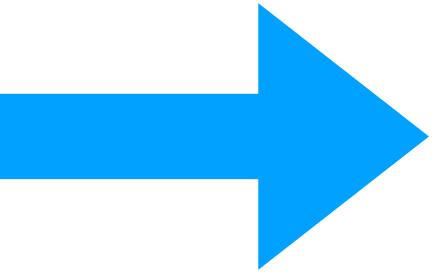
primrec rev1::"'a list ⇒ 'a list" where
  "rev1 []      = []"
  | "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2 []      ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
  apply auto done

 $\exists r1 : \text{rule}. \text{True}$ 
 $\rightarrow$ 
 $\exists r1 : \text{rule}.$ 
 $\exists t1 : \text{term}.$ 
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$ 
 $r1 \text{ is\_rule\_of } to1$ 
 $\wedge$ 
 $\forall t2 : \text{term} \in \text{induction\_term}.$ 
 $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$ 
 $\exists n : \text{number}.$ 
 $\text{is\_nth\_argument\_of } (to2, n, to1)$ 
 $\wedge$ 
 $t2 \text{ is\_nth\_induction\_term } n$ 

```



True



```

primrec rev1::"'a list  $\Rightarrow$  'a list" where
  "rev1 [] = []"
  | "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2:: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list" where
  "rev2 [] ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev xs @ ys"
  apply(induct ys xs rule: rev2.induct)
  apply auto done

 $\exists r1 : \text{rule}. \text{True}$ 
 $\rightarrow$ 
 $\exists r1 : \text{rule}.$ 
 $\exists t1 : \text{term}.$ 
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$ 
 $r1 \text{ is\_rule\_of } to1$ 
 $\wedge$ 
 $\forall t2 : \text{term} \in \text{induction\_term}.$ 
 $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$ 
 $\exists n : \text{number}.$ 
 $\text{is\_nth\_argument\_of } (to2, n, to1)$ 
 $\wedge$ 
 $t2 \text{ is\_nth\_induction\_term } n$ 

```

```

primrec rev1::"'a list  $\Rightarrow$  'a list" where
  "rev1 [] = []"
  | "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2:: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list" where
  "rev2 [] ys = ys"
  | "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev xs @ ys"
  apply(induct ys xs rule: rev2.induct)
  apply auto done

```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$



False



Candidate induction?



Candidate induction?



Can't you tell me what arguments to pass to the induct tactic?

Candidate induction?



Can't you tell me what arguments to pass to the induct tactic?

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



smart_induct

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

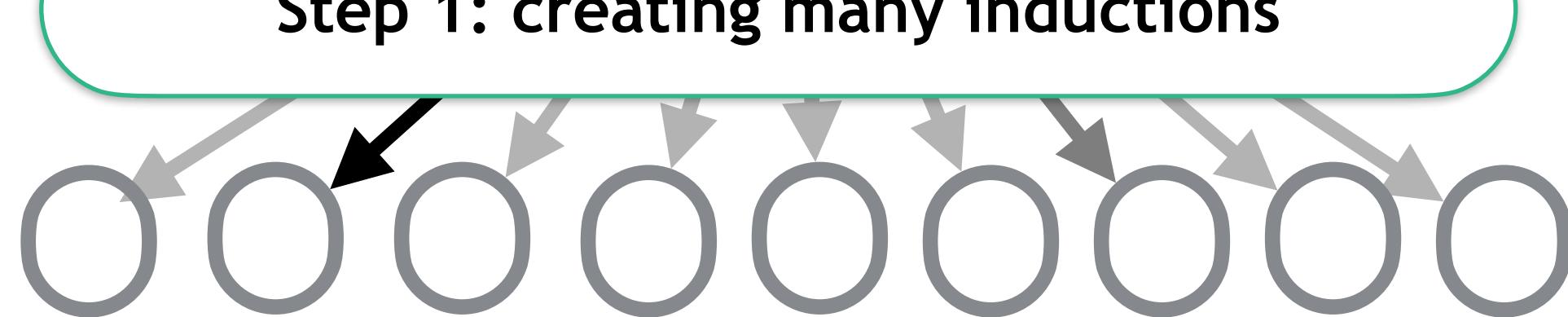
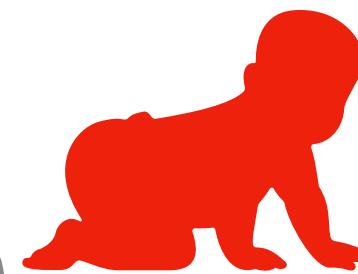


goal

smart_induct

Step 1: creating many inductions

new users



Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



goal

smart_induct

Step 1: creating many inductions

new users



Step 2: filtering out unpromising tactics

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

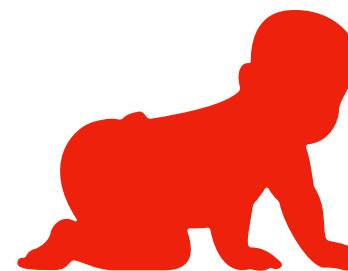


goal

smart_induct

Step 1: creating many inductions

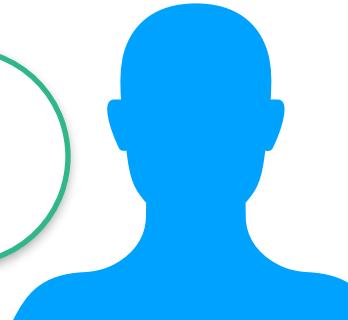
new users



Step 2: filtering out unpromising tactics

experts

Step 3: scoring using 19 heuristics and sorting



20 20 19 18 18 18 17

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

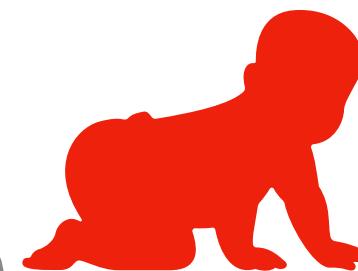


goal

smart_induct

Step 1: creating many inductions

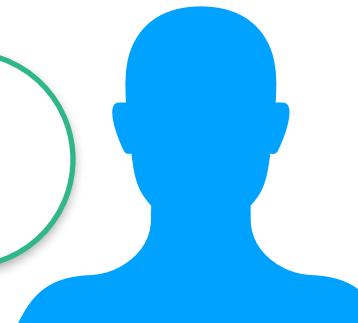
new users



Step 2: filtering out unpromising tactics

experts

Step 3: scoring using 19 heuristics and sorting



20 20 19 18 18 18 17

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

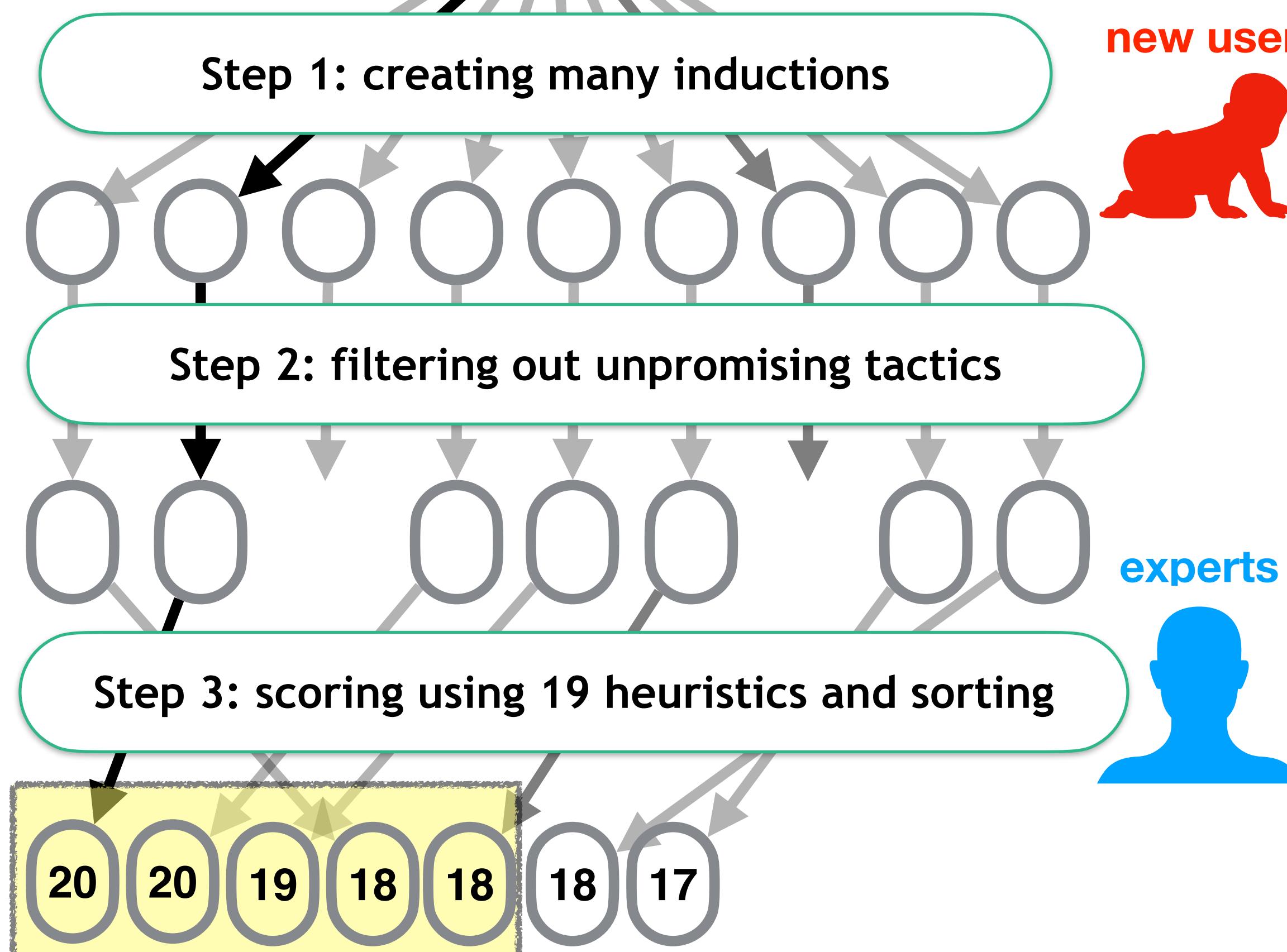
Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

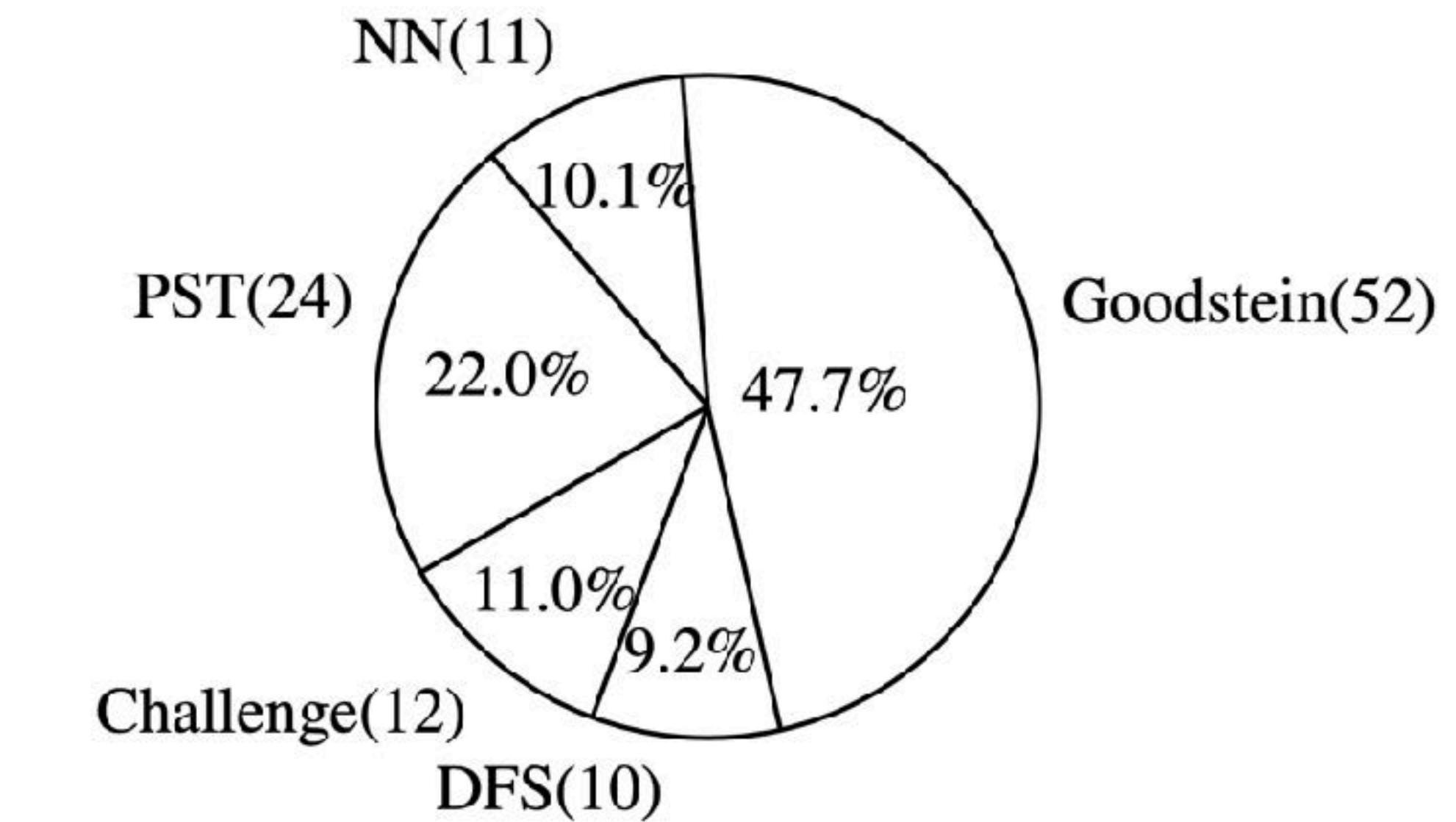


goal

smart_induct



Dataset for evaluation
(109 proofs by induction from the AFP entries)



Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

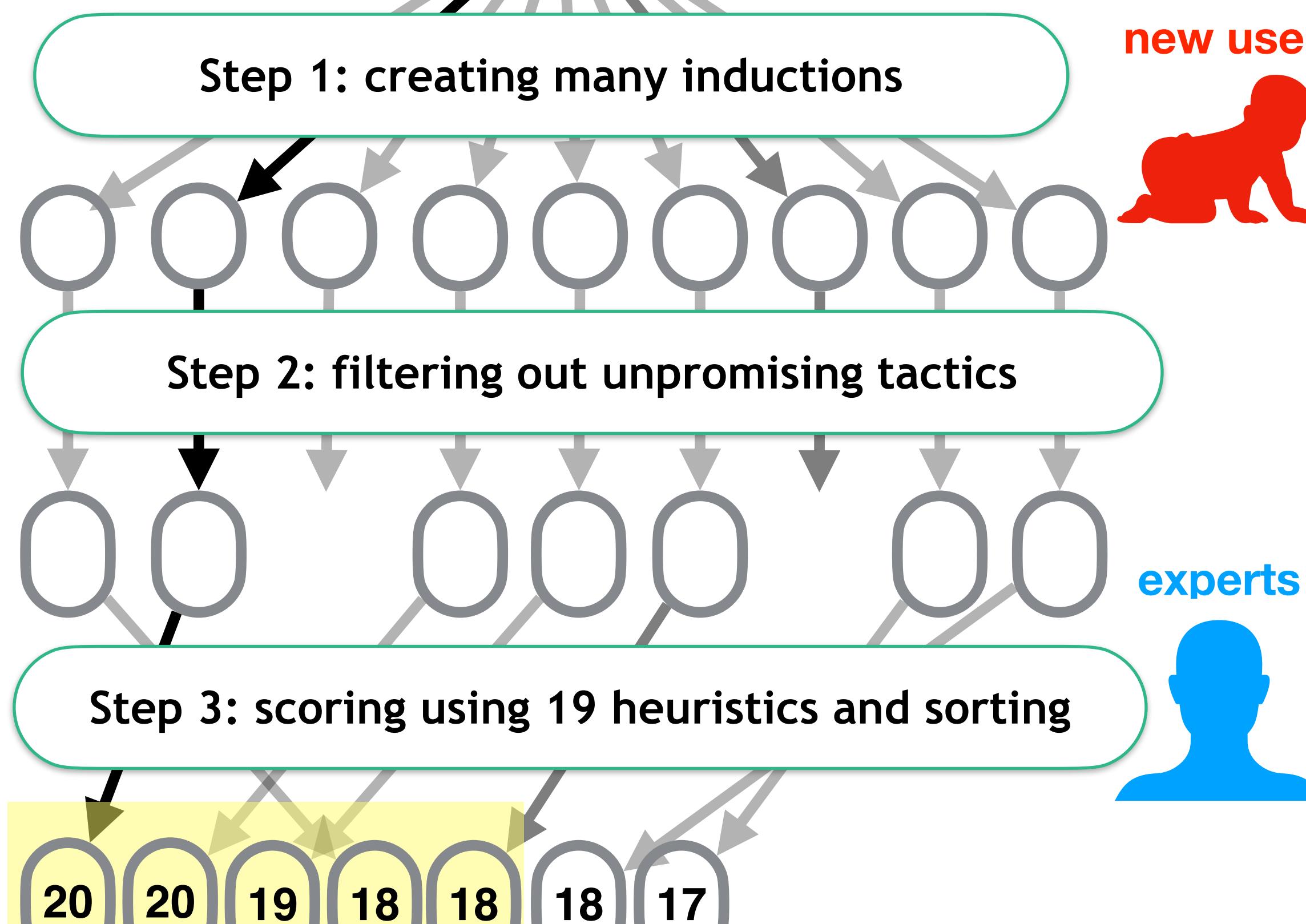
Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

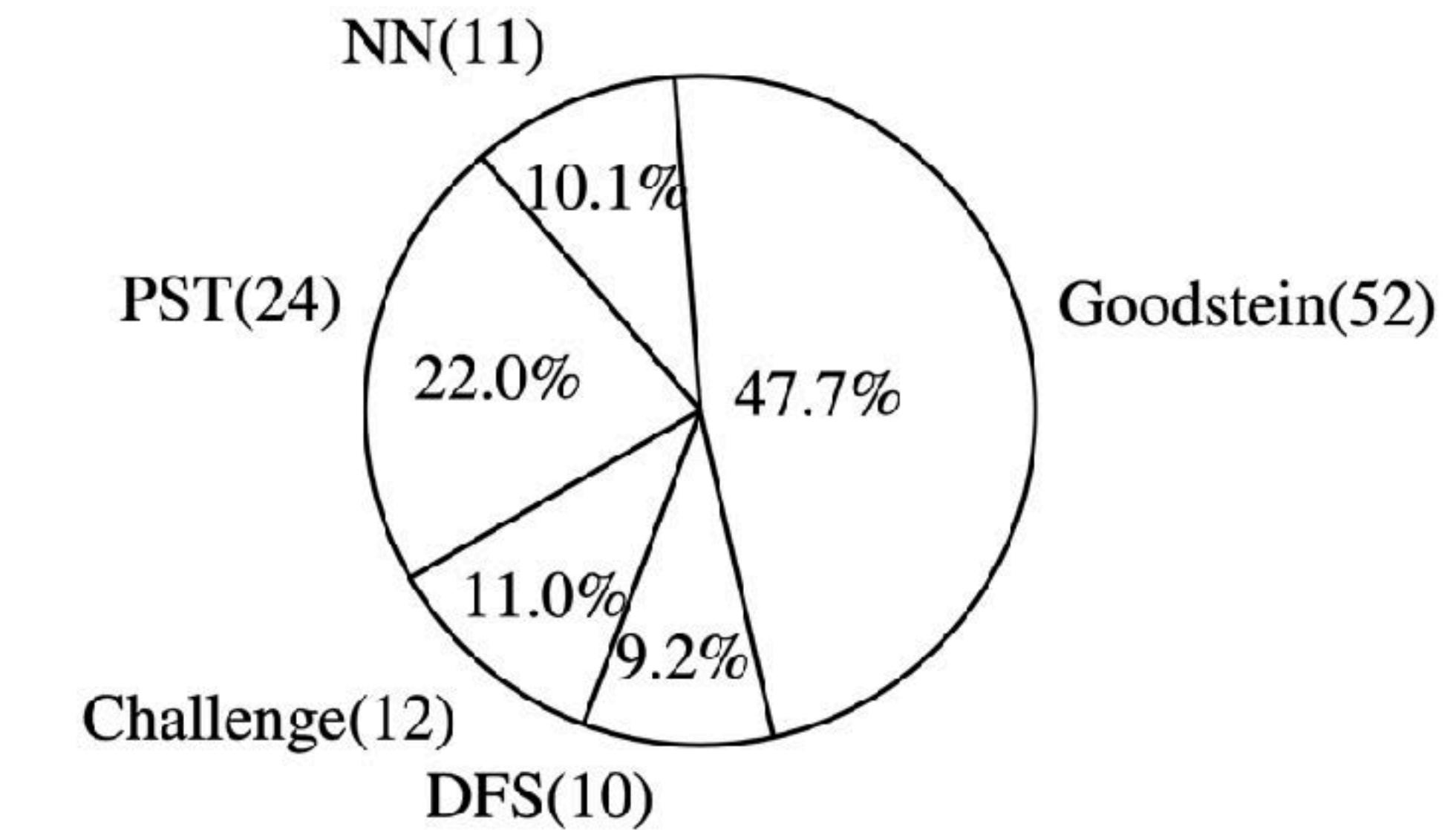


goal

smart_induct



Dataset for evaluation
(109 proofs by induction from the AFP entries)



Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



Algorithmic approaches work well, but not always.

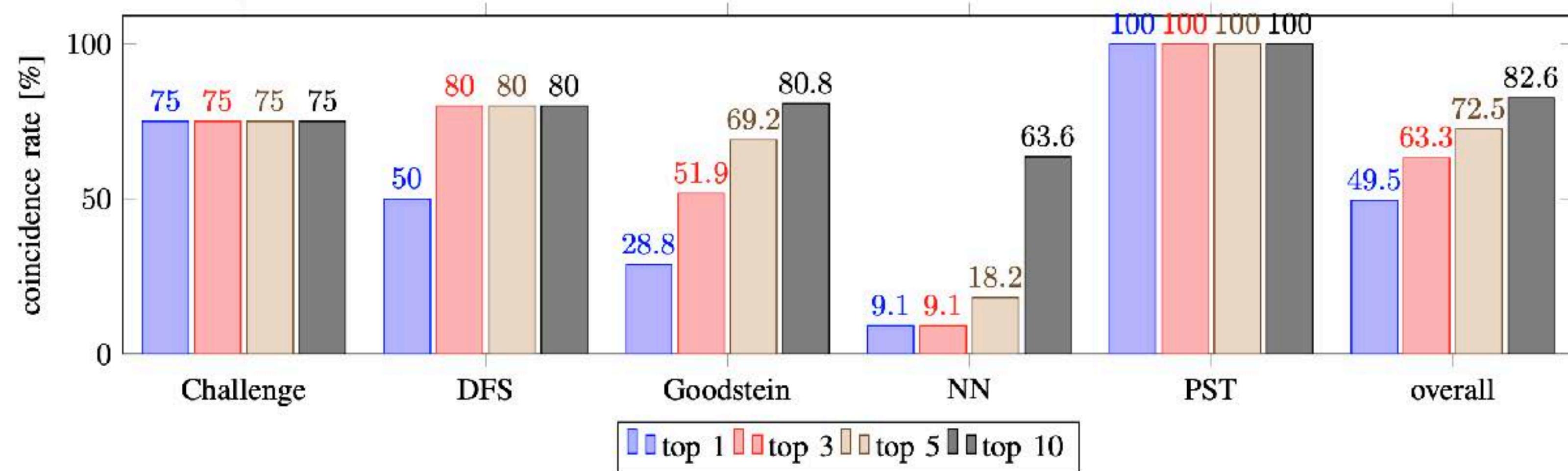
There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



Coincidence rates



Algorithmic approaches work well, but not always.

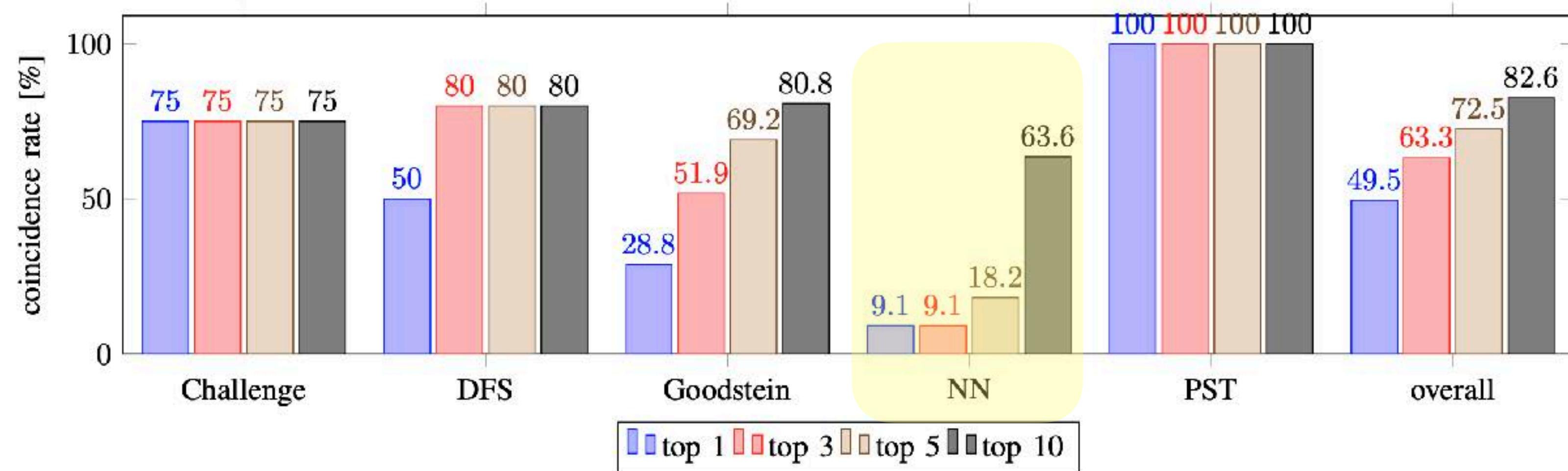
There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.



Coincidence rates



Algorithmic approaches work well, but not always.

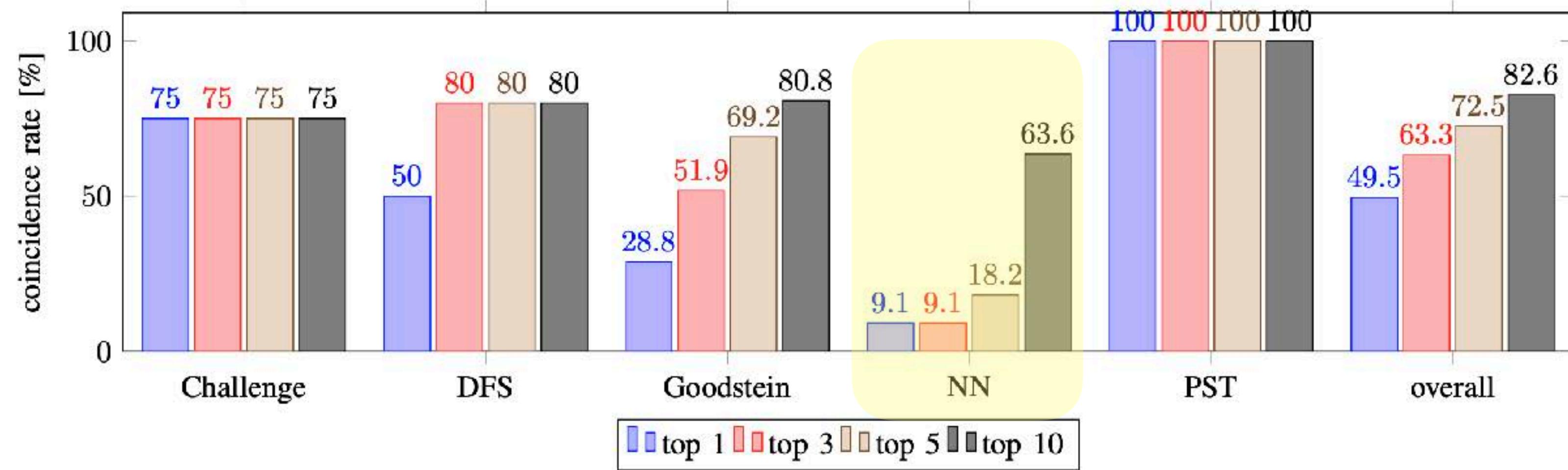
There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

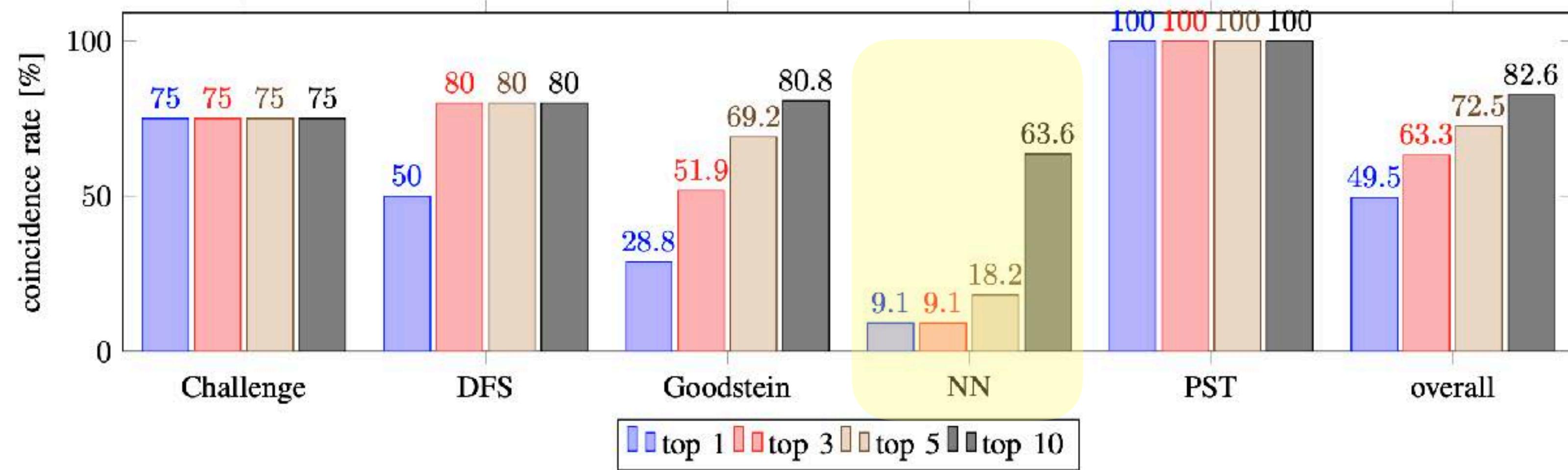
It is okay to make mistakes if we can identify them quickly.



Coincidence rates

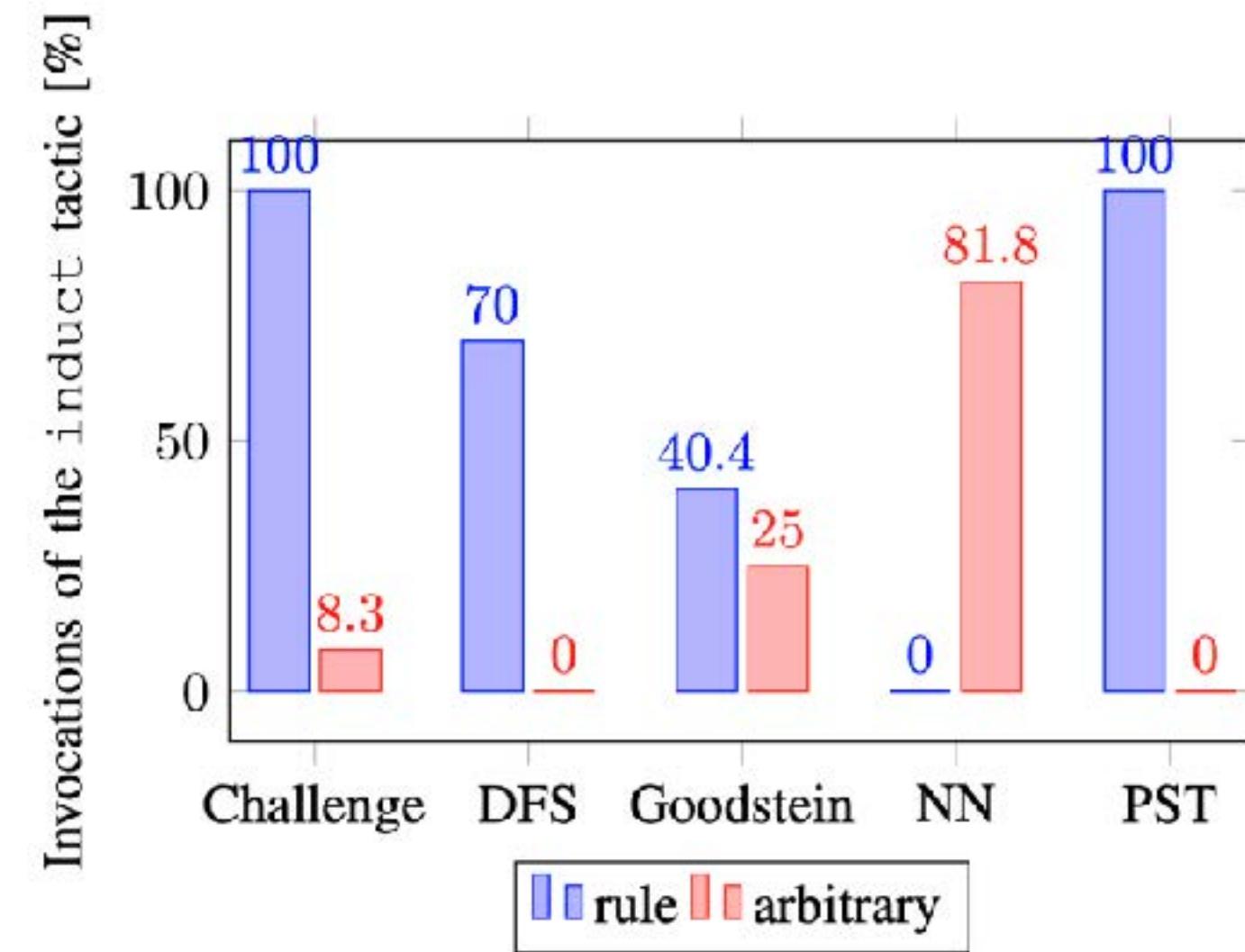
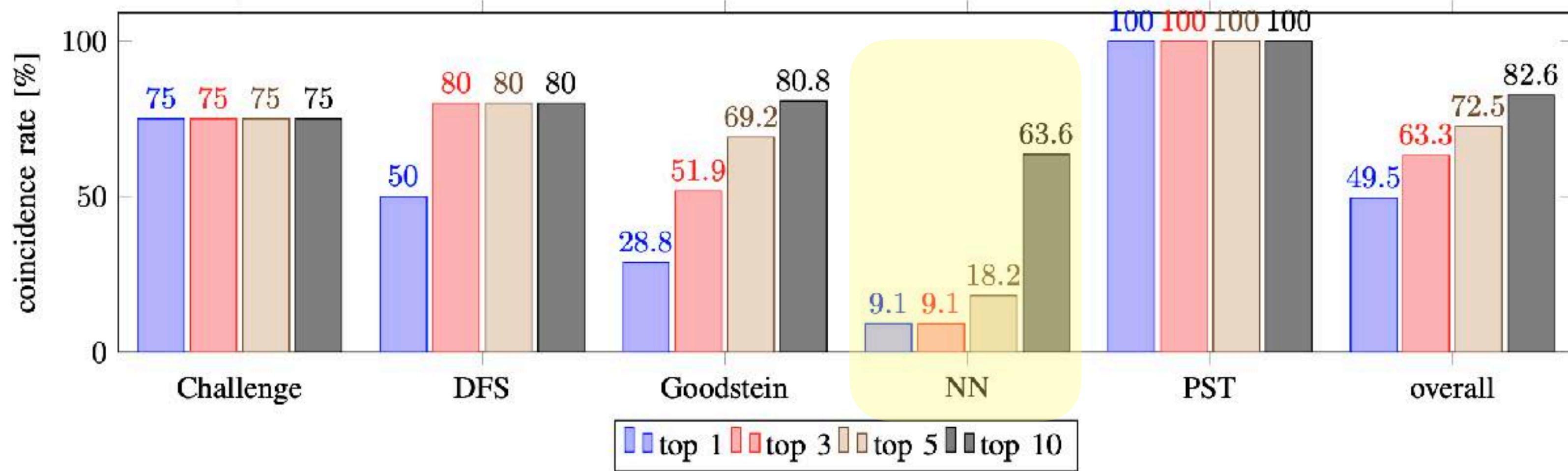


Coincidence rates



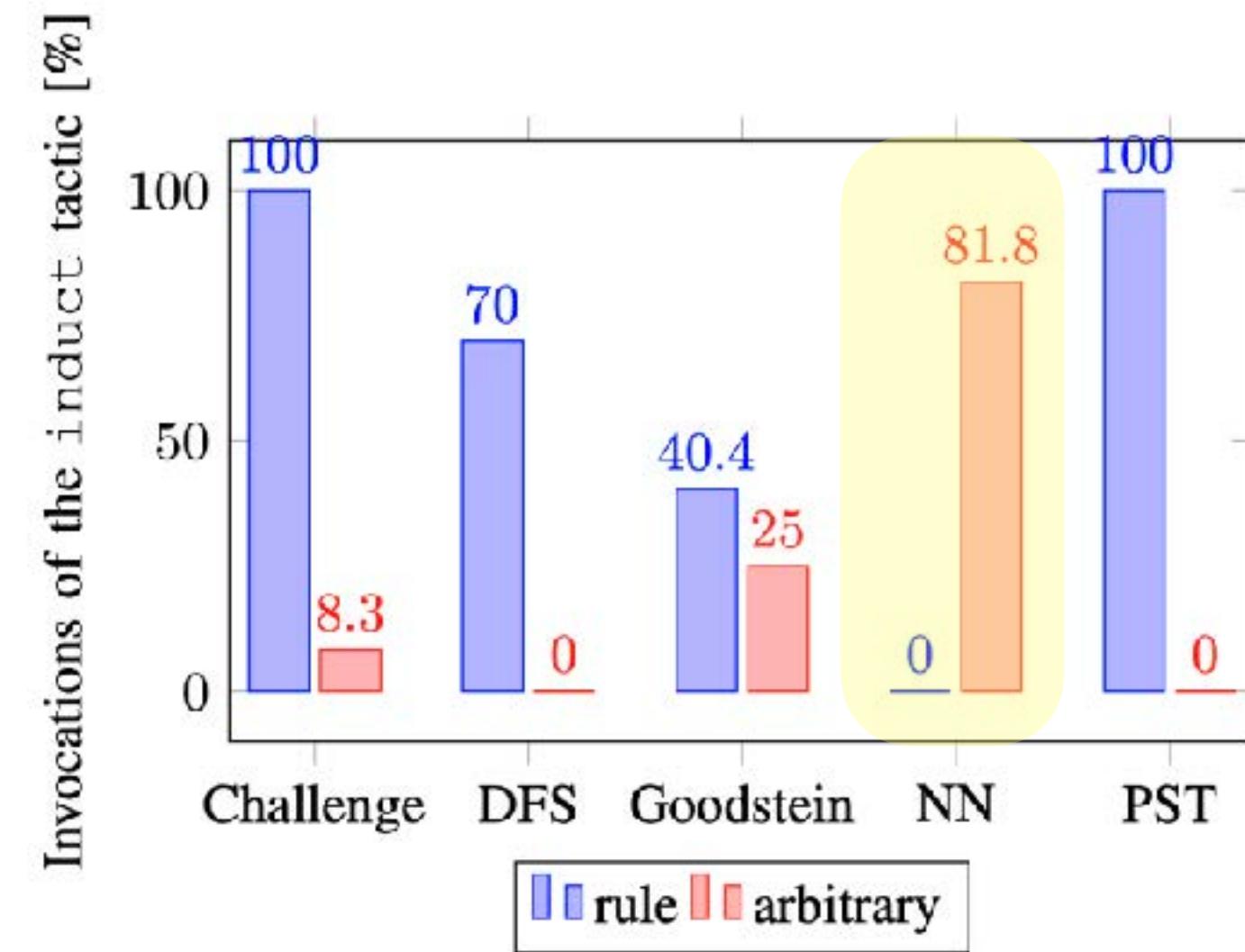
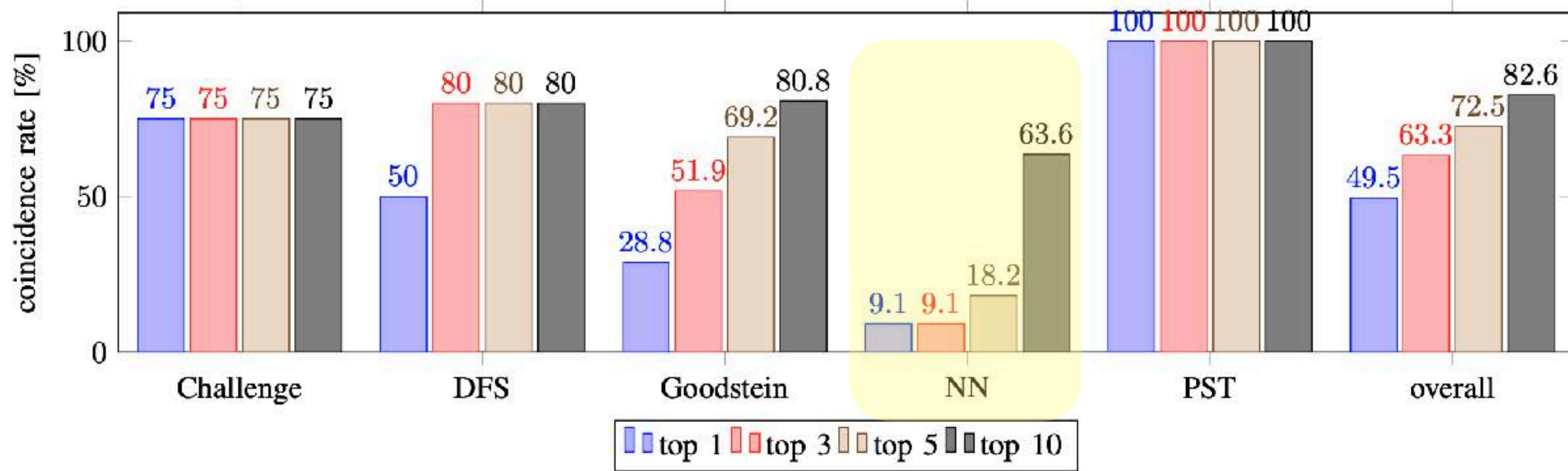
Is smart_induct Domain dependent?

Coincidence rates



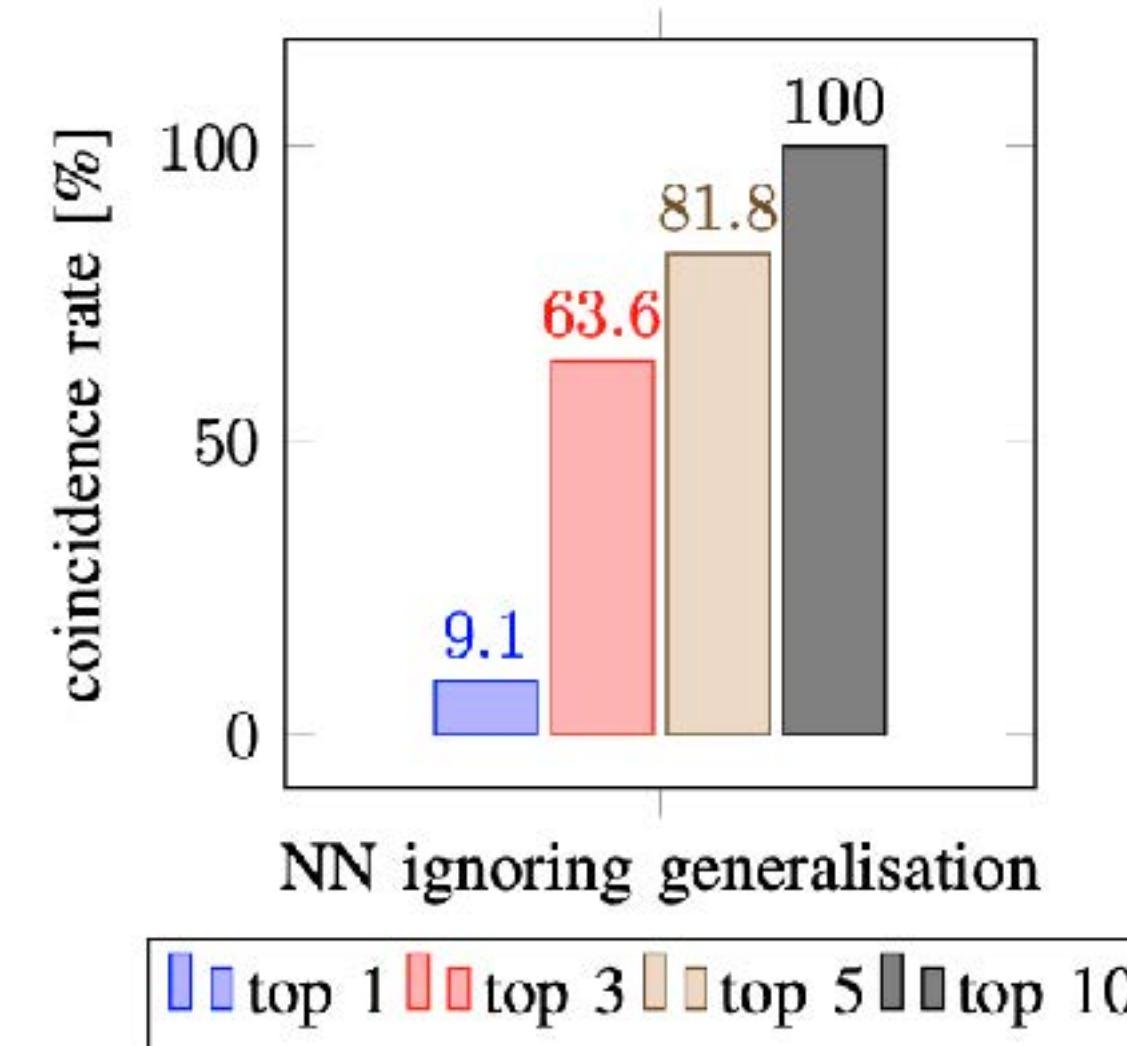
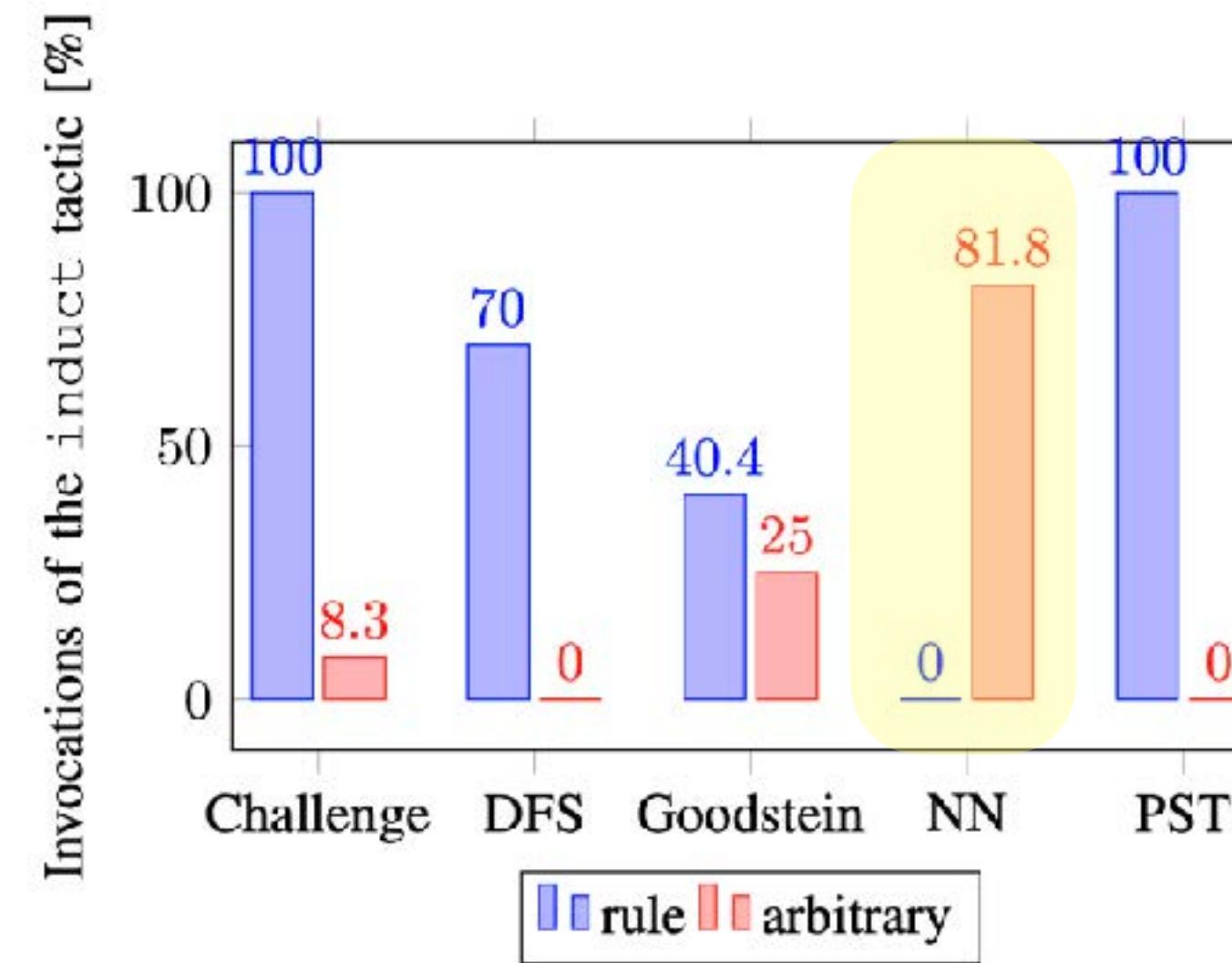
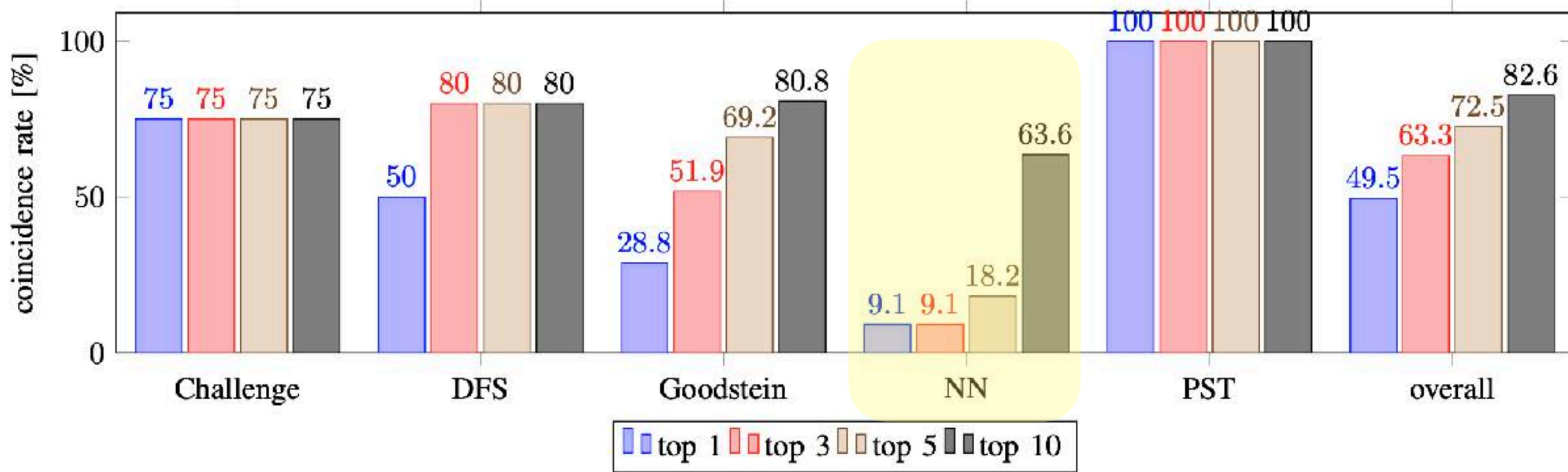
Is smart_induct Domain dependent?

Coincidence rates



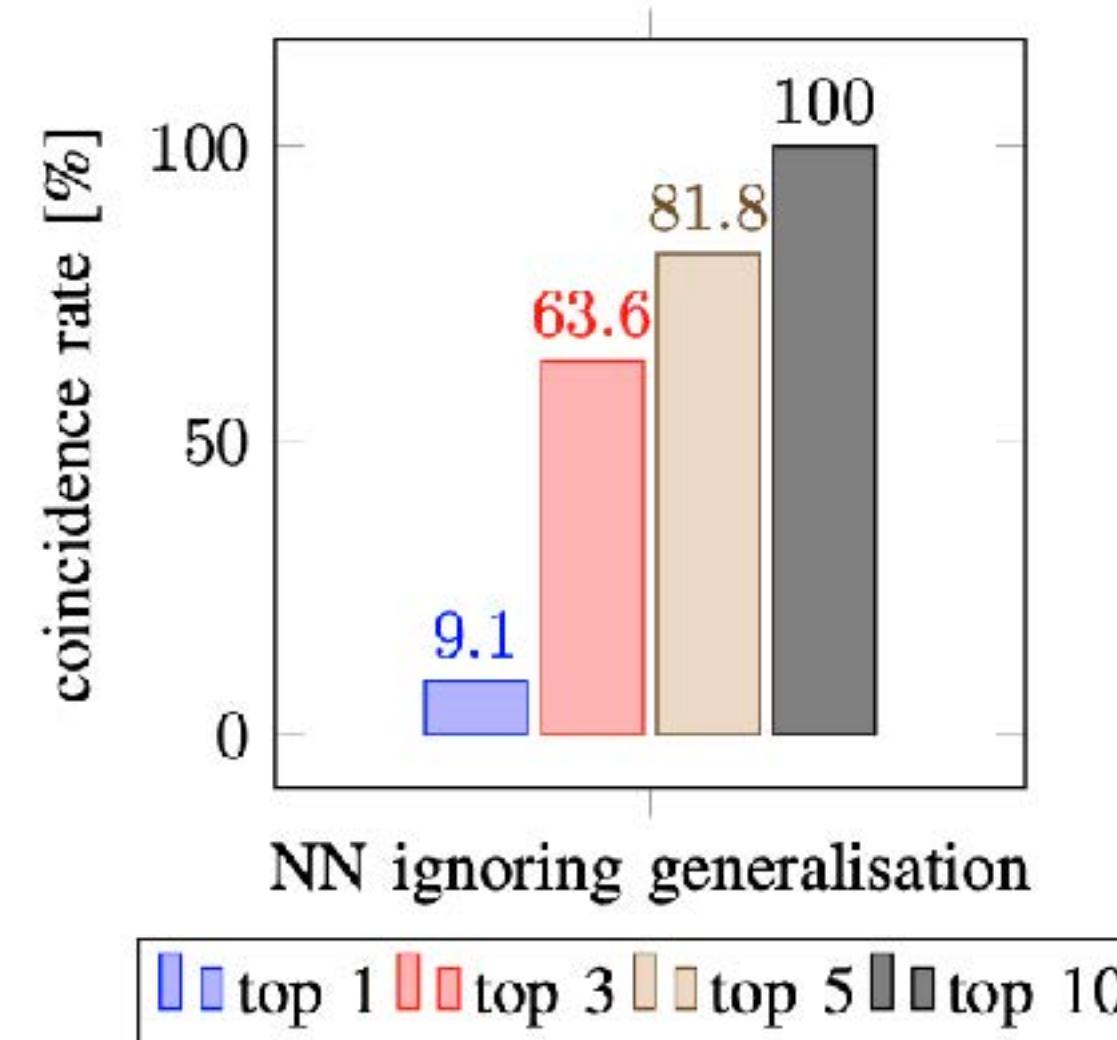
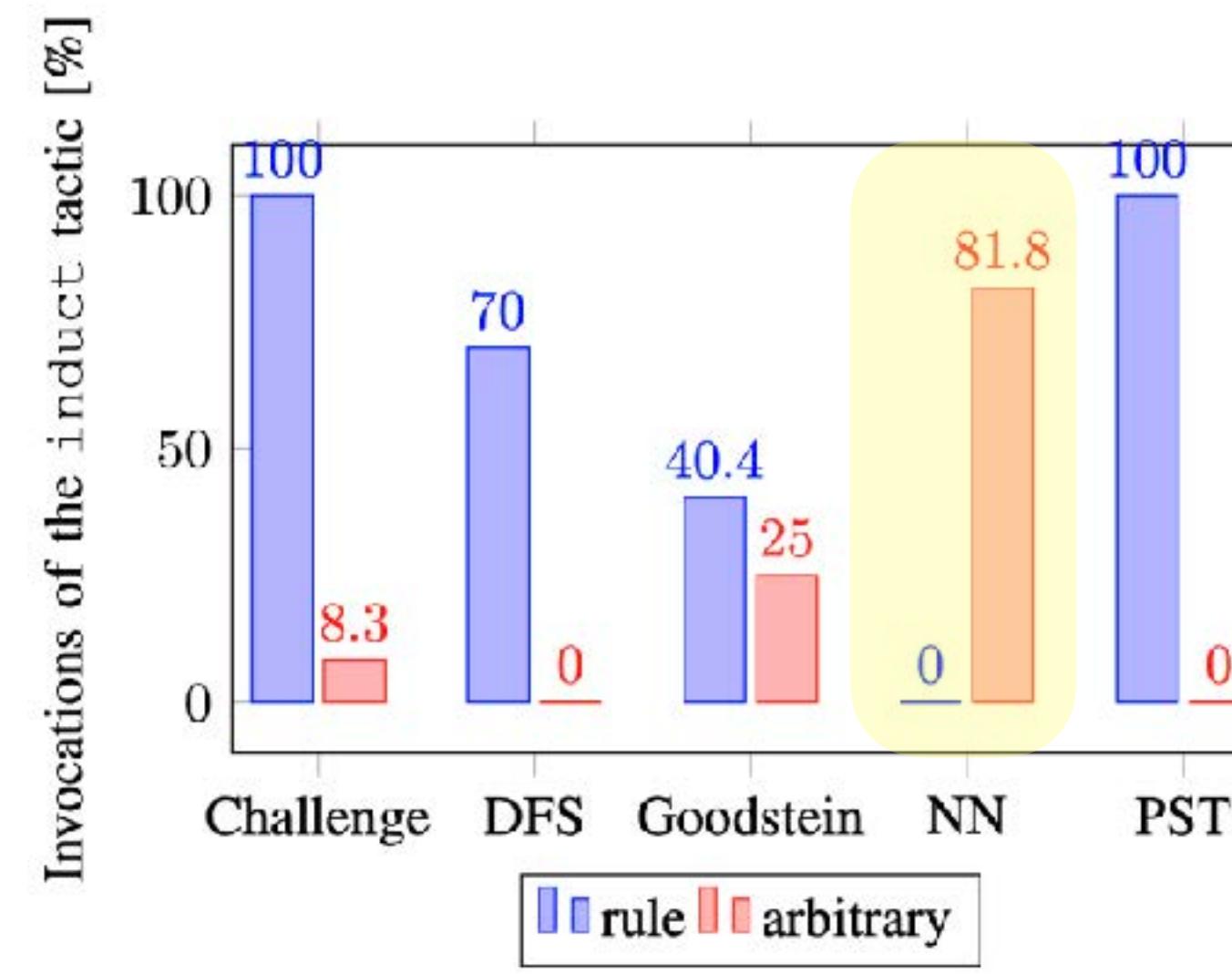
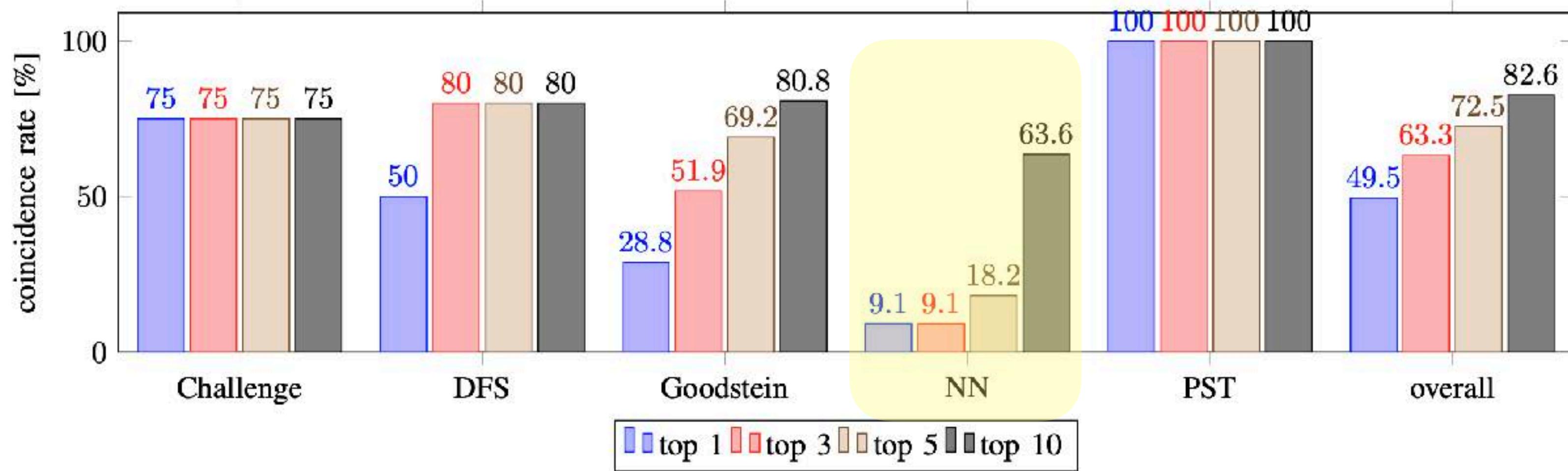
Is smart_induct Domain dependent?

Coincidence rates



Is smart_induct Domain dependent?

Coincidence rates



Is smart_induct Domain dependent?

smart_induct is bad at generalisation heuristics.



Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).

b: Logical Feature Extraction (LiFtEr).

c: smart_induct using LiFtEr.

Step 3: Putting things together.



Program 4 Automatic inductive prover without SeLFiE

```
strategy Auto_Solve = Thens [Auto, IsSolved]
strategy PSL_WO_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Dynamic (Induct), Auto_Solve]
    PThenOne [
      Dynamic (Induct),
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 5 Automatic inductive prover with SeLFiE

```
strategy PSL_W_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Semantic_Induct, Auto_Solve]
    PThenOne [
      Semantic_Induct,
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 4 Automatic inductive prover without SeLFiE

```
strategy Auto_Solve = Thens [Auto, IsSolved]
strategy PSL_WO_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Dynamic (Induct), Auto_Solve]
    PThenOne [
      Dynamic (Induct),
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 5 Automatic inductive prover with SeLFiE

```
strategy PSL_W_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Semantic_Induct, Auto_Solve]
    PThenOne [
      Semantic_Induct,
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Success rate

| timeouts | Program 5 | Program 4 |
|----------|-----------|-----------|
| 0.3[s] | 11.0% | 1.2% |
| 1.0[s] | 25.6% | 1.7% |
| 3.0[s] | 28.2% | 21.9% |
| 10.0[s] | 34.9% | 28.0% |
| 30.0[s] | 45.8% | 38.3% |

Bibliography

Step 1: Tactic recommendation.

<https://doi.org/10.1145/3238147.3238210>

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).

https://doi.org/10.1007/978-3-319-63046-5_32

b: Logical Feature Extraction (LiFEr).

https://doi.org/10.1007/978-3-030-34175-6_14

c: smart_induct using LiFEr.

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_32

d: Semantic-aware Logical Feature Extraction (SeLFE).

<https://arxiv.org/pdf/2010.10296.pdf>

e: sem_ind using SeLFE.

<https://arxiv.org/pdf/2009.09215.pdf>

f: conjecturing using abductive-reasoning.

https://doi.org/10.1007/978-3-319-96812-4_19



fin!

